BIBLIOTECA BASICA INFORMATICA

LISP 

un nuevo lenguaje

INGELEK

BIBLIOTECA BASICA INFORMATICA

LISP 32

un nuevo lenguaje

INGELEK

Director editor:
Antonio M. Ferrer Abelló.

Director de producción:
Vicente Robles.

Coordinador y supervisión técnica:
Enrique Monsalve.

Redactores técnicos:
Juan José Alba
Jesús Pedraza
José Miguel Pinilla

Redactor técnico:
Alfredo B. García Pérez.

Colaboradores:
Casimiro Zaragoza.

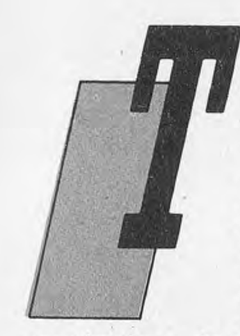
Diseño:
Bravo/Lofish.

Dibujos:
José Ochoa.

ISBN del tomo: 84-85831-73-X
ISBN de la obra: 84-85831-31-4
Fotocomposición Pérez Díaz, S. A.
Imprime: Héroes, S. A.
Depósito Legal: M-16.946-1986
Precio en Canarias, Ceuta y Melilla: 380 pts.

INDICE

PROLOGO



Tradicionalmente se tiende a pensar en los lenguajes de programación como una forma más o menos sofisticada de dar instrucciones a un ordenador. Sin embargo, son algo más: cada lenguaje lleva asociado un determinado modelo de la máquina. Aunque se utilice siempre el mismo equipo material, basta emplear lenguajes diferentes para conferir al ordenador distintas personalidades, puesto que el programador piensa en términos de variables, no de celdas de memoria; en ficheros de datos y no en dispositivos de entrada y salida, o en fórmulas en lugar de en registros y sumadores.

El objetivo de este libro es que el lector obtenga una idea clara de las posibilidades de aplicación de LISP y adquiera conocimientos suficientes para entender los programas escritos en este lenguaje, así como para poder confeccionar los suyos propios.

El tratamiento de la estructura del lenguaje no pretende ser exhaustivo ni excesivamente profundo, para facilitar la comprensión del mismo al lector novel. Sin embargo, se ha intentado mantener en todo momento el rigor en los conceptos expresados, de modo que esta lectura sirva como base sólida para una futura profundización.

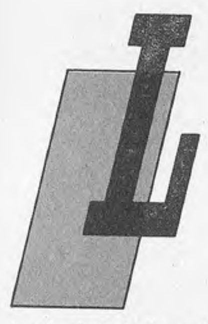
LISP es uno de los lenguajes más simples que se pueden concebir. Su uso induce al programador habituado a los más conocidos lenguajes algebraicos a desarrollar un concepto enteramente nuevo de la estructura de un programa.

Este es el primer lenguaje que suministra una estructura básica sencilla con unas amplias posibilidades de extensión. Ha sido aplicado a problemas que implican manipulación de símbolos y

a desarrollos en el campo de la Inteligencia Artificial. Esto es debido, por una parte, a su gran facilidad para el tratamiento de símbolos, y por otra, a que su estructura interactiva se adapta bien a la tendencia de los programadores de Inteligencia Artificial a ser perezosos e indisciplinados, como viajeros que rechazan trazarse un recorrido antes de dirigirse a parte alguna.

CAPITULO I

LISP: INTRODUCCION



ISP fue desarrollado a final de los años cincuenta por John McCarthy, del Instituto Tecnológico de Massachusetts (MIT), que originalmente creó una versión primitiva del mismo bastante difícil de leer por un ser humano. Poco después lo modificó para desarrollar un programa matemático que requería que tanto los procedimientos como los datos tuvieran la misma sintaxis. El resultado fue algo bastante parecido a la forma actual del lenguaje. Su nombre deriva de **LISt Proccesing,** puesto que tanto los programas como los datos se estructuran en forma de listas.

El lector debe saber que el LISP no ha sido normalizado de la misma forma que otros lenguajes de programación, como pueda ser el FORTRAN. En este libro se utiliza el dialecto conocido como **Common Lisp,** uno de los más recientes y extendidos. Al final haremos algunos comentarios sobre otros dialectos y daremos unas indicaciones sobre la forma de compatibilizarlos entre sí.

Cabe destacar que, aunque asociado tradicionalmente a la Inteligencia Artificial, el lenguaje LISP tiene ciertas características que facilitan la tarea del programador en aplicaciones muy variadas. Por ejemplo, en áreas como el diseño de circuitos integrados VLSI, el proceso de señales digitales, la enseñanza de la informática, el desarrollo de sistemas informáticos (como sistemas operativos, compiladores, intérpretes y programas de utilidades diversas), o el manejo de símbolos matemáticos, los entornos de trabajo LISP pueden llegar a competir (o compiten ya actualmente) en eficiencia con los sistemas basados en PASCAL o FORTRAN.

Para el usuario de pequeños ordenadores personales hay en

el mercado mini-sistemas LISP de alta calidad, adecuados para aprender el lenguaje y escribir programas sencillos. Sin embargo, estas versiones están limitadas por la escasez de memoria y el alto consumo de tiempo de proceso. Con el desarrollo de ordenadores más avanzados podrán paliarse estos defectos; entretanto, con la lectura de los restantes capítulos se podrá aprender a manejar los sistemas LISP disponibles hasta el momento y estar preparado para la tecnología que vendrá en un futuro próximo.

Antes de comenzar a describir el lenguaje LISP, conviene hacer una serie de comentarios sobre la formalización de los procesos y el concepto de funciones y datos. El lector experimentado puede omitirlos y continuar la lectura en el capítulo 2.

Formalización

El razonamiento formal es un método de gran importancia para tratar el mundo que nos rodea, puesto que constituye la base de la ciencia y las matemáticas modernas.

Al tratar con los ordenadores debemos indicarles con precisión qué es lo que deben hacer. Esto lo hacemos con los programas, que no son sino la descripción de una tarea.

Prácticamente todas las tareas, entendiendo como tales cualquier actividad desarrollada por un ente capaz de influir sobre su entorno (ser humano, animal), pueden ser formalizadas y, por tanto, simuladas por un programa. Aquí no está incluido el tipo de razonamiento que nos lleva a escribir o interpretar una poesía, o a sentir temor o necesidad de algo, por ejemplo.

Un ejemplo de formalización de una acción es el proceso de adquisición de un ordenador personal:

- Determinar de nuestras necesidades.
- Dirigirnos a la tienda de ordenadores más cercana.
- Pedir información sobre los modelos existentes en el mercado.
- Seleccionar aquellos cuyas prestaciones satisfagan nuestras necesidades presentes y futuras.
- Elegir entre este grupo aquel que ofrezca mejores condiciones económicas.

Este proceso tiene un nivel de detalle suficiente como para que una persona mínimamente familiarizada con los ordenadores pueda seguirlo. Sin embargo, es algo demasiado vago para un ordenador. Este no sabe, por ejemplo, qué es una tienda de ordenadores o qué criterio de selección debe seguirse en la valoración de las prestaciones.

Al trabajar en LISP se tiene la importante ventaja sobre otros lenguajes de programación de poder tratar con facilidad símbolos que representan conceptos, como ordenador o modelos de ordenadores. Sin embargo, sigue presente el problema básico: debemos describir de forma precisa y completa el procedimiento que debe seguirse, teniendo en cuenta las posibles incidencias, como puede ser en el caso anterior que la tienda esté cerrada o que no podamos pagar ninguno de los ordenadores seleccionados, por ejemplo.

Funciones y datos

En cualquier lenguaje, pero especialmente en LISP, se utilizan de forma constante los conceptos de función y de datos.

En general, los datos son manipulados por las funciones para obtener nuevos datos. Una forma muy ilustrativa de representar esto es considerar la función como una fábrica que dispone de todos los contenidos necesarios para realizar un proceso a determinado tipo de datos. Por ejemplo, la fábrica "PADRE-DE" tiene acceso a toda la información necesaria para hallar el nombre del padre de la persona cuyo nombre se suministra como dato de entrada; este ejemplo lo tenemos representado en la figura 1. Al introducir el nombre Juan en la fábrica se obtiene como producto el nombre del padre de Juan, que es Pedro.

Conviene aquí hacer una puntualización: podemos considerar la fábrica a dos niveles: un nivel material, al cual corresponde el proceso físico de transformación, y un nivel administrativo, al que corresponde el proceso burocrático de la misma. Con un ejemplo se aclararán estas dos ideas: llega un pedido a una fábrica de clavos; por ejemplo, una caja de bobinas de alambre. En pri-

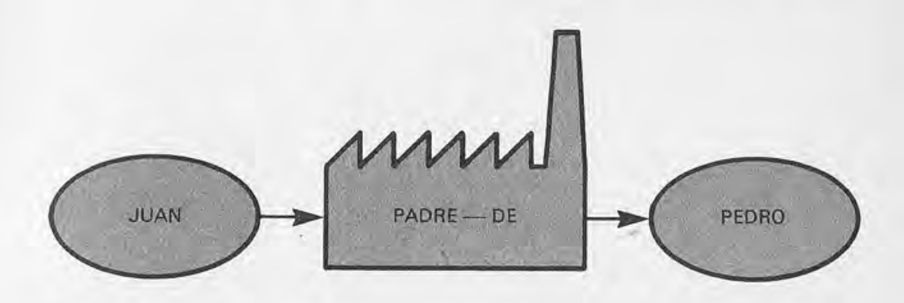


Figura 1.—El padre de Juan es Pedro.

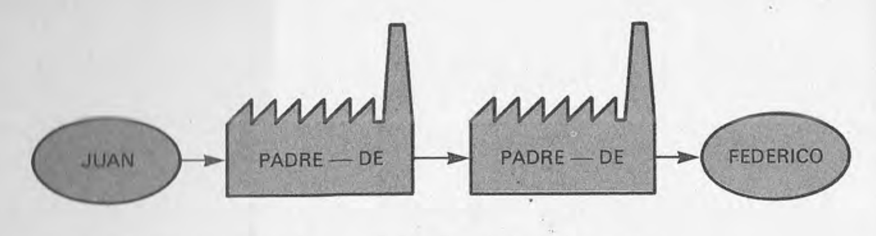


Figura 2.—El padre del padre de Juan es Federico.

mer lugar éste es identificado a través del albarán en donde se detalla el contenido del mismo; una vez hecho esto se lleva a la sección correspondiente, donde sufre una serie de transformaciones. Al final de ellas se obtienen los clavos, que se ven reflejados en el nivel administrativo como una nota en la que se especifican sus características.

Pues bien, las funciones en LISP realizan un proceso análogo. Reciben unos datos de entrada (el albarán), que normalmente son usados para representar a los objetos con los que trabajan (las bobinas de alambre), y devuelven el resultado (los clavos) asociado a unos datos de salida que lo representan (la nota de características).

Al igual que los productos elaborados por una fábrica pueden ser transformados por otra (la bobina de alambre fue el producto elaborado por otra fábrica a partir del metal correspondiente, por ejemplo), las funciones pueden encadenarse para realizar transformaciones sucesivas de los datos (Fig. 2). Esto viene a ser equivalente a definir una nueva función que haga todo el proceso seguido (Fig. 3) y posteriormente aplicársela a los datos de entrada (Fig. 4).

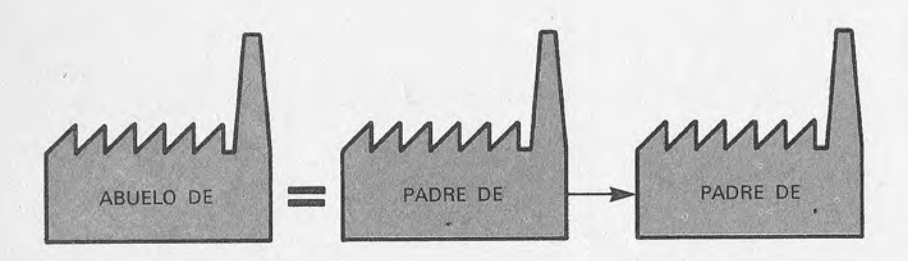


Figura 3.—El padre del padre de una persona es su abuelo.

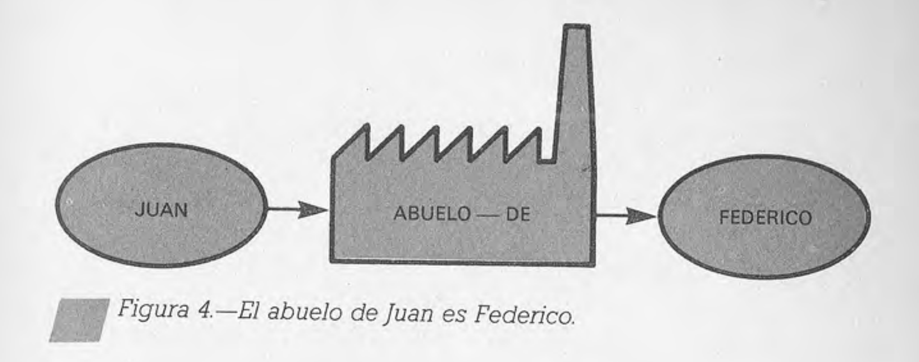


Figura 4.—El abuelo de Juan es Federico.

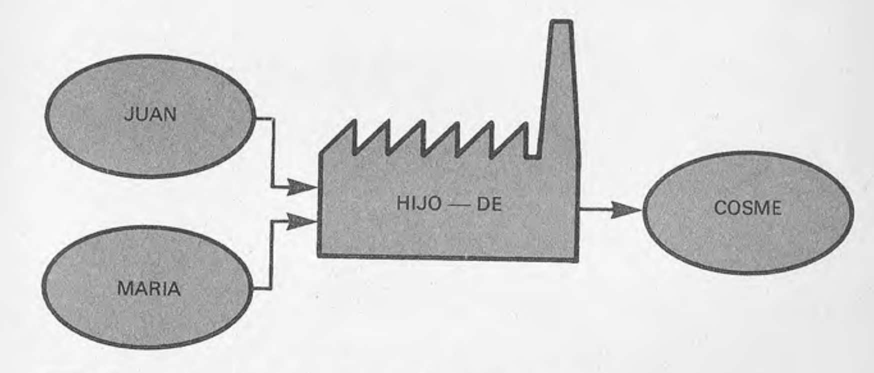


Figura 5.—El hijo de Juan y María es Cosme.

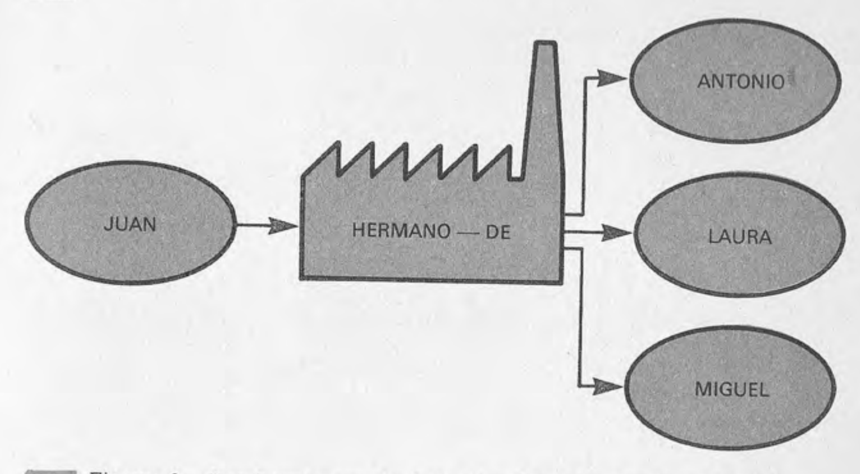


Figura 6.—Los hermanos de Juan son Antonio, Laura y Miguel.

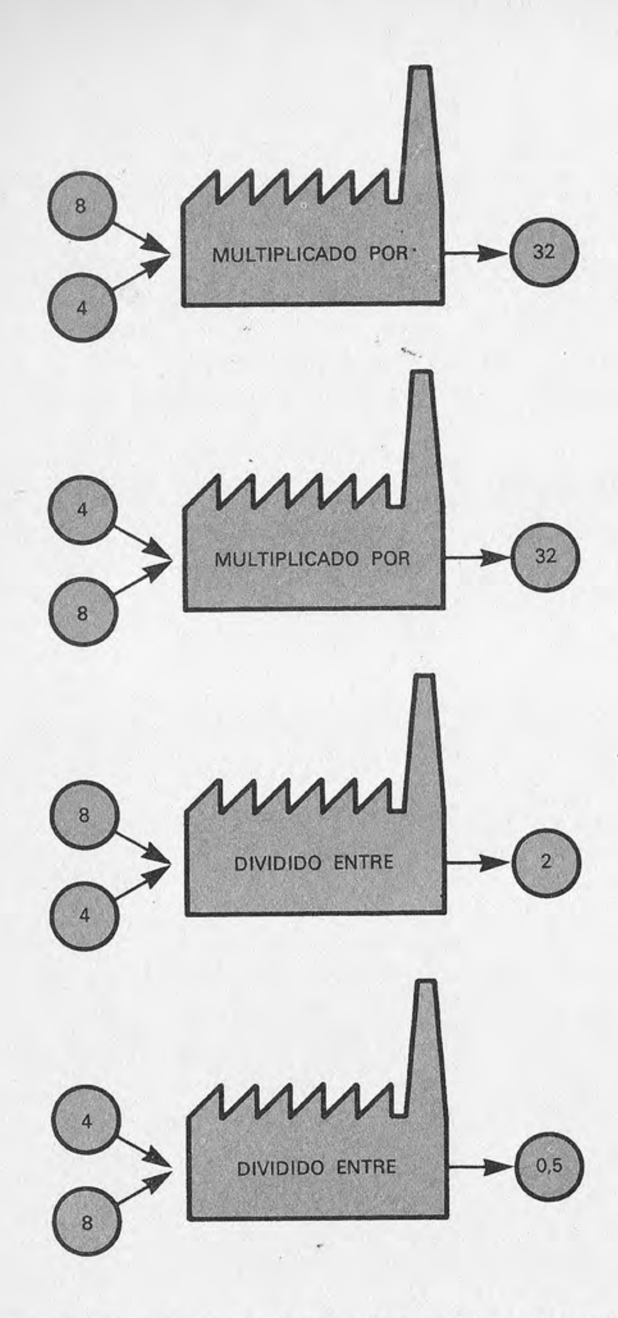


Figura 7.—El orden de los datos es importante en determinadas funciones.

Las funciones pueden tener varios argumentos de entrada (Fig. 5) o de salida (Fig. 6). En algunos casos el orden en que se suministren estos argumentos es indiferente y en otros es muy importante. Las funciones multiplicación y división son un ejemplo de cada uno de estos tipos de funciones (Fig. 7).

Las funciones que dan como valor de salida los valores cierto o falso se denominan predicados. Se utilizan para comprobar si los datos de entrada cumplen unos requisitos. En la figura 8 hay un ejemplo de predicado en el que el orden de los datos de entrada es importante, puesto que el primero de ellos es el hijo y los dos siguientes son los padres.

Los predicados pueden combinarse con otras funciones para formar nuevos predicados. Por ejemplo, SOBRINO-DE?, puede construirse a partir de HIJO-DE y HERMANO-DE (Fig. 9).

En LISP hay muchos tipos diferentes de datos y es posible trabajar con funciones adecuadas a cada uno de ellos. A lo largo de este libro se introducirán paulatinamente las funciones más importantes, con numerosos ejemplos para su mejor comprensión.

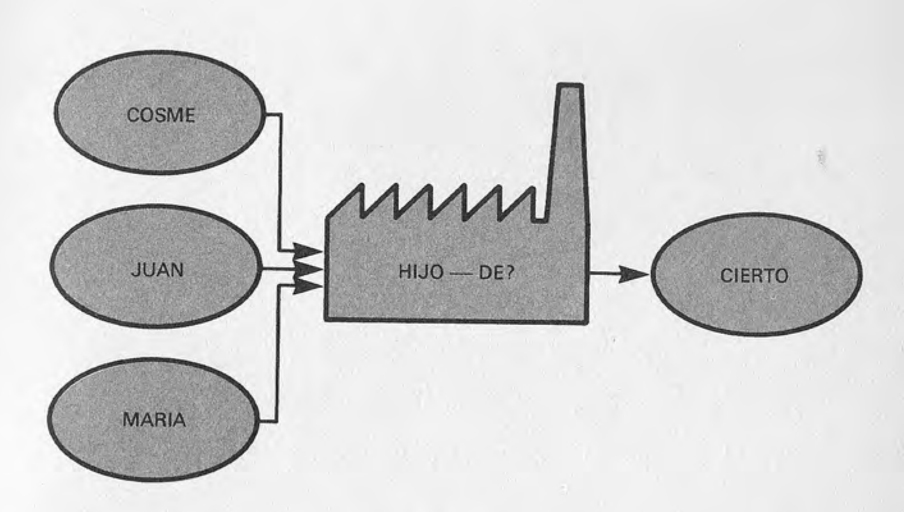


Figura 8.—Cosme es hijo de Juan y María.

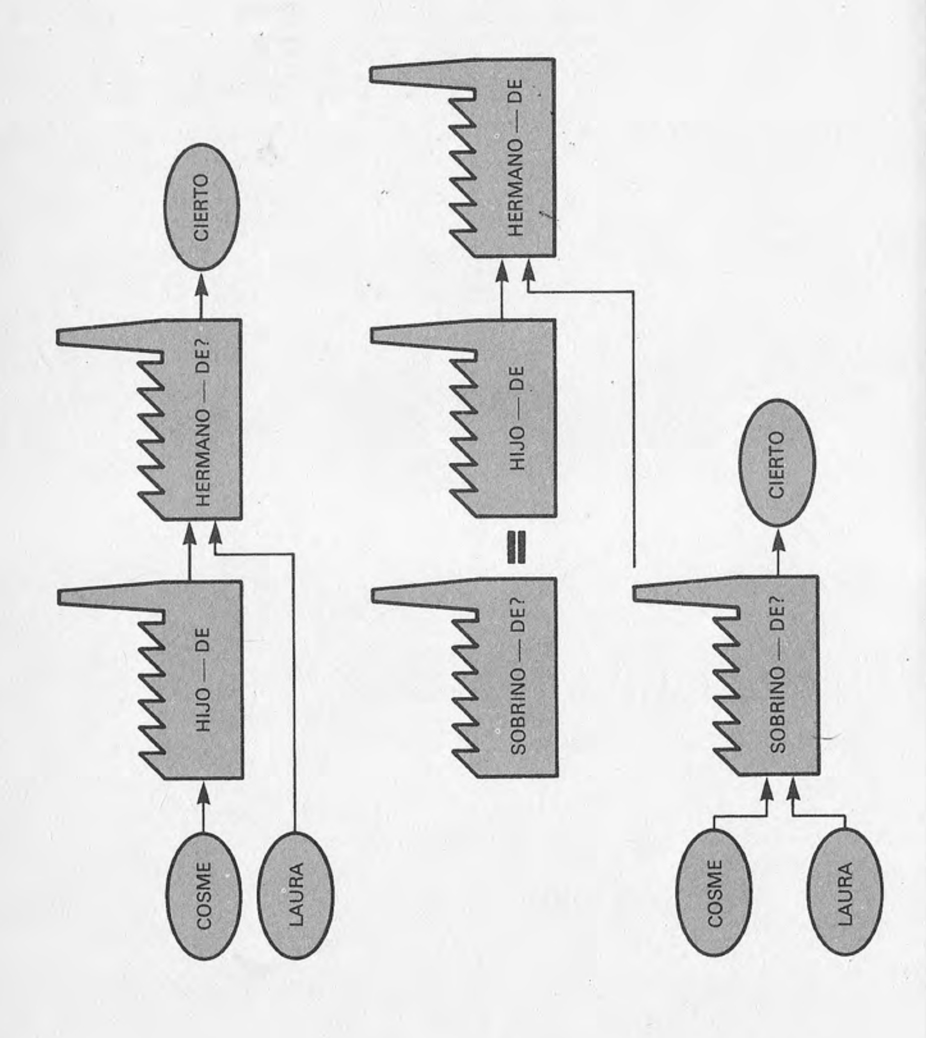
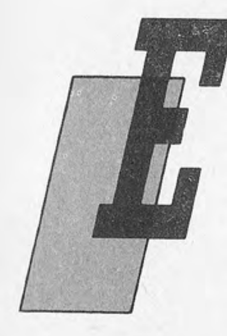


Figura 9.—Cosme es hijo de Juan, que es hermano de Laura.

CAPITULO II

TIPOS BASICOS DE DATOS



EN LISP, una correcta comprensión de los tipos de datos es imprescindible para poder utilizar toda la potencia del lenguaje. Los tipos básicos dentro del variado repertorio de LISP son: números, símbolos, cadenas de caracteres, listas, matrices, estructuras y funciones.

El concepto de *expresión* engloba a todos los tipos y es la unidad básica de programa en LISP.

Números

El LISP soporta varios tipos de números, como son:

- **Enteros** (ej. 0, –10, 56, 123456, ...)
- **Coma flotante o decimales** (ej. 0.1, 10.27, 0.1258, ...). De éstos existen varias longitudes para proveer la adecuada precisión o eficiencia en los cálculos.
- **Complejos** (ej. (2,3), (5.1,2), ...). Representados en forma cartesiana; sus partes imaginaria y real pueden ser de los tipos anteriores, aunque no de distinto tipo entre sí.

Símbolos

Un símbolo es un conjunto de caracteres que empieza por una letra o un número y NO es un número. Ejemplos de éstos son:

FOO, BAR, BAZ, 1+, +$, PE-PI-TO. Sin embargo, +1, +7, -18.5 no son símbolos.

Los caracteres permitidos son las letras, los números y, además, los siguientes:

=, -, *, /, @, $, %, ^, &, -, \, <, >, ~, .

Los símbolos tienen una serie de características que los hacen muy interesantes. Pueden nombrar o representar objetos que, a su vez, se distingan por sus peculiares propiedades y que se pueden asociar al símbolo.

Un símbolo puede representar cualquier tipo de objeto en LISP, incluso otro símbolo.

Existen dos símbolos reservados en Common LISP, que son "T" y "NIL". Representan los valores de certeza y falsedad, respectivamente.

Cadenas de caracteres

Están formadas por letras, números y otros caracteres especiales, como %, espacio, etc. Se representan entre comillas.

Listas

Son, con mucho, las estructuras más importantes del LISP. Consisten en un paréntesis, seguido de una serie de objetos LISP, acabada en un paréntesis de cierre. Ejemplos de listas son:

```
(1  2  3  4)
(1  (2 3)  4)
(FOO BAR)
(BAZ)
( )
(1 FOO (2 3) (BAR (T BAZ)))
```

La lista vacía se representa por "NIL" o por "( )"; así "NIL" es el único objeto que es símbolo y lista al mismo tiempo.

Una definición más rigurosa de lista es la siguiente. Una lista es:

a) la lista vacía o NIL.
b) el resultado de añadir una expresión cualquiera a la cabeza de una lista.

Esta es una definición *recursiva*, capaz de generar cualquier lista. Por ejemplo, la lista (A (B 1) C 17) se formará como sigue:



1. Lista vacía .... ( )
2. 17 & ( ) .... (17)
3. C & (17) .... (C 17)
4. (B 1) & (C 17) .... ((B 1) C 17)
5. A & ((B 1) C 17) .... (A (B 1) C 17)

Las listas en LISP tienen una representación interna muy interesante, que es la que les confiere sus características principales. La unidad básica de construcción de una lista es lo que se denomina **célula elemental.** Consiste en dos punteros asociados entre sí, y la representaremos como se indica en la figura 1.

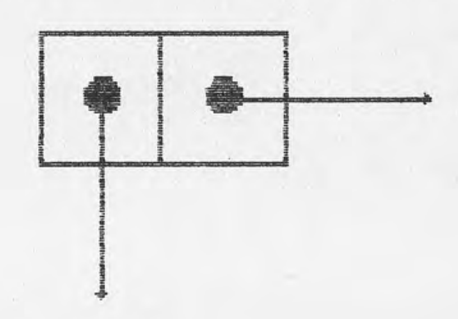


Figura 1.—Una célula elemental.

Estos dos punteros tienen nombre propio: CAR el primero y CDR el segundo. Bajo esta estructura las listas son una serie de células elementales encadenadas y que terminan en NIL. Así, la lista (A B) tiene la estructura que se muestra en la figura 2. En la figura 3 se dan algunos ejemplos de listas y sus representaciones.

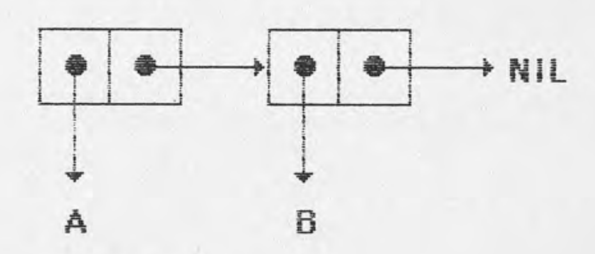


Figura 2.—La lista (A B).

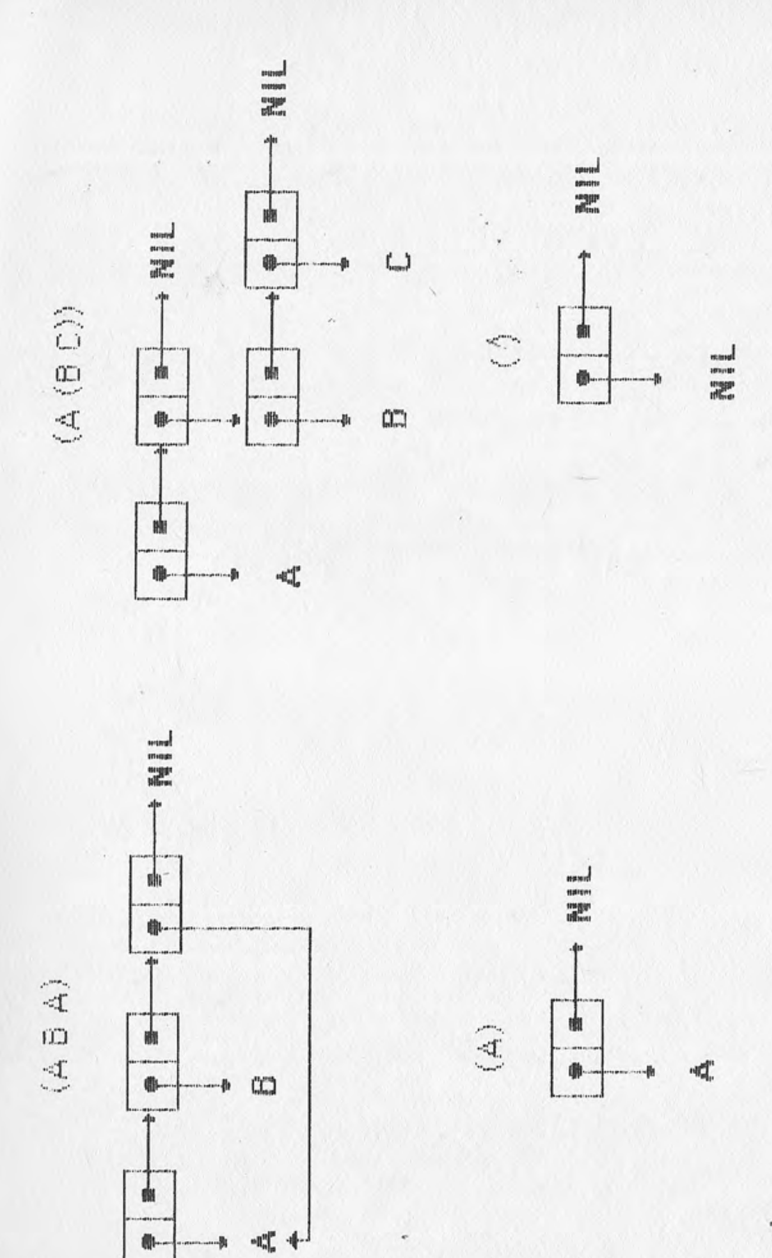
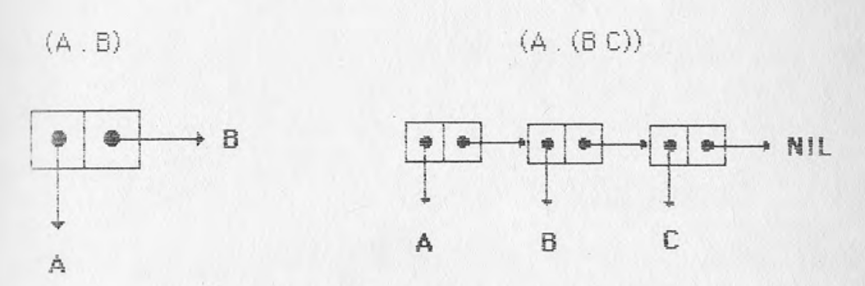


Figura 3.—Algunas listas y su representación celular.

Aunque en general todas las listas terminan en NIL, cabe pensar en una estructura análoga que no lo haga. Es lo que se denomina **par ordenado**. Se representa por un paréntesis, un objeto LISP, un punto, otro objeto LISP y el cierre del paréntesis. En la figura 4 hay algunos ejemplos.



Esta estructura, nunca se verá (A . (B C)), sino así: (A B C)

Es interesante comparar estas estructuras con las de las figuras 2 y 3.

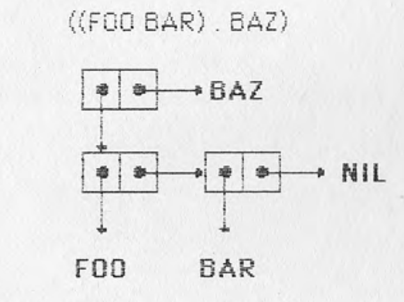


Figura 4.—Algunos pares ordenados y su representación celular.

Matrices

Corresponden a las matrices matemáticas. Sus elementos pueden ser de los diversos tipos que admite el LISP, incluso pueden existir varios tipos en una misma matriz.

Estructuras

Las estructuras son construcciones que deben ser definidas por el usuario; almacenan datos de forma semejante a como lo hacen los registros de una base de datos, es decir, con una serie de campos en cada registro. Al definir una estructura, el intérprete LISP crea una serie de funciones que permiten acceder y modificar la información almacenada en ella.

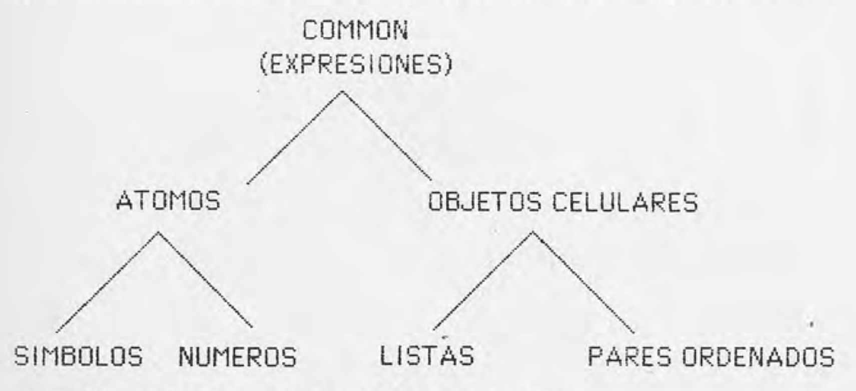
NOTA.—A lo largo del libro hacemos uso de la palabra "estructura" con dos sentidos. En el capítulo 11 se refiere al tipo de datos así llamado y que acabamos de comentar por encima, mientras que en otros capítulos podrá encontrar la referencia a "estructuras de datos", con lo que denominaremos a todo tipo de datos.

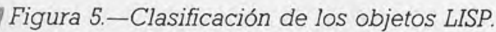
Funciones

Son los principales procedimientos que permiten manipular los datos. Las hay específicas de cada clase de datos, como, por ejemplo, las aritméticas, funciones con listas, etc.

Todas estas estructuras de datos y otras que aquí no hemos mencionado se clasifican en grupos jerárquicamente organizados. La clase de todas las posibles estructuras válidas en Common LISP se denomina **common.**

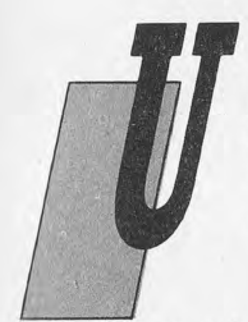
De todas las estructuras se dice que son **expresiones.** Otras clases importantes son los **átomos** (símbolos y números principalmente) y sus opuestos (las listas y los pares ordenados), genéricamente llamados **objetos celulares.** La característica que los diferencia consiste en estar formados por células elementales (objetos celulares) o no (átomos). La figura 5 ilustra esta clasificación.




Figura 5.—Clasificación de los objetos LISP.

CAPITULO III

ESTRUCTURA DE LOS PROGRAMAS: SINTAXIS Y SEMANTICA



na vez que se han definido los tipos de datos más importantes, la sintaxis del LISP se hace de una simpleza extraordinaria y se resume en una frase:

CUALQUIER EXPRESION CORRECTA EN LISP ES UN PROGRAMA O PROCEDIMIENTO SINTACTICAMENTE CORRECTO

Como ya se ha comentado, una expresión puede ser, típicamente, un átomo o una lista. Un átomo siempre es una expresión correcta, en términos de sintaxis. Una lista será sintácticamente correcta si cada paréntesis derecho se corresponde con un paréntesis izquierdo. Por tanto, programas LISP pueden ser:

```
53

FOO

(ADD1 5)

(CAR (CDR '(BAR BAZ )))

(EL-PADRE-ES 'FEDERICO)

(PERRO TIENE 4 PATAS)

NIL
```

Para tratar la semántica LISP es esencial comprender lo que se entiende por **evaluar** una expresión. En LISP todas las expre-

siones tienen algún valor asignado, así la **evaluación** de una expresión consiste en obtener su valor.

El LISP trabaja con un bucle denominado READ-EVAL-PRINT (Lectura-Evaluación-Impresión), que toma una expresión, bien sea del teclado o de otro dispositivo (ficheros, memoria, etc), obtiene su valor y lo muestra, bien sea en pantalla o en otro dispositivo. Puede ilustrarse esto con un ejemplo. si se teclea:

(+ 2 5)

el intérprete LISP lo leerá, interpretará y después imprimirá:

7

Las reglas de evaluación son bastante sencillas y se dan a continuación. A partir de ahora, el resultado de evaluar una expresión se presentará separado de ésta por una flecha.

a) Los números evalúan su valor.

```
 7    ->     7
 5    ->     5
 1.25  ->    1.25
-17.0  ->   -17.0
```

b) NIL y T se evalúan a sí mismos

```
T    ->     T
NIL  ->     NIL
```

c) Los símbolos evalúan la expresión que les ha sido asociada

```
PADRE-DE-JUAN  ------->    FEDERICO
PERRO          ------->    ANIMAL
FOO            ------->    (BAR BAZ)
X              ------->    7
FALSO          ------->    NIL
```

Es de notar que a los símbolos se les puede asociar cualquier expresión, sea otro símbolo, una lista, un número o cualquier otra, siempre que sea correcta.

d) La evaluación de las listas es algo más compleja y da al LISP su carácter. El LISP asume que la primera expresión es una forma especial, una macro o una función, produciéndose error en otro caso. El tipo de esta expresión determina la evaluación de la lista.

Evaluación de listas

Si el primer elemento de la lista es una **función**, el LISP considera al resto de las expresiones de la lista como argumentos de la misma. Los evalúa y después los manipula según el procedimiento que determine la función. Si se tienen dos funciones: "+", que suma todos sus argumentos, y "*", que multiplica los suyos, pueden plantearse algunos ejemplos.

```
(+ 2 4)    ->   6
(* 10 6)   ->   60
```

La evaluación de los argumentos se muestra a continuación.

```
   (* (+ 7 3) (+ 2 4))  ->
-> (*   10       6   )  ->   60
```

Por supuesto, pueden anidarse tantas listas como se desee:

```
(* 10 (* 9 (* 8 (+ 5 (+ 2 7))))) -> 10080
```

Si se tiene un símbolo al que se le asignó un valor, por ejemplo asignamos 12 a "docenas", entonces

```
   (* docenas (+ 5 2)) ->
-> (*   12       7  )  ->   84
```

Los ejemplos que presentamos en el capítulo se escribirán en LISP como sigue:

(PADRE-DE 'JUAN) → PEDRO
(PADRE-DE (PADRE-DE 'JUAN)) → FEDERICO
(HIJO-DE 'JUAN 'MARIA) → COSME

Nótese el apóstrofe que aparece justo antes de los símbolos "JUAN" y "MARIA". Aunque es un elemento importante del LISP lo trataremos algo más adelante.

La primera expresión de una lista también puede ser una forma especial o el nombre de macro.

Las **formas especiales** son procedimientos de manipulación de datos y control que trabajan de manera particular. Pueden o no evaluar sus argumentos y los efectos que tienen son propios de cada una.

En Common LISP hay un número fijo de formas especiales, ya preprogramadas, y a las que no se puede añadir ninguna creada por el usuario. A continuación se explica brevemente el comportamiento de una forma especial como ilustración.

La forma especial SETQ sirve para dar un valor a un símbolo. Su sintaxis es como sigue:

(SETQ símbolo expresión)

SETQ no evalúa el símbolo, pero sí la expresión, asignando a aquél el valor retornado por ésta. Además, la lista evalúa este valor. Por ejemplo:

(SETQ DOCENA (* 6 2)) → 12
DOCENA → 12

En el teclado se escribe la expresión, que aparece al mismo tiempo en pantalla. Al pulsar RETORNO es evaluada y devuelve su valor: 12. A continuación se teclea DOCENA y, al evaluarlo, nos da su valor: 12, que le ha sido asignado en la primera forma.

Una **macro** maneja expresiones; al ejecutarse construye una nueva forma, que es la que se evaluará. Este proceso se denomina **expansión.** Cuando aparece el nombre de una macro en la cabeza de una lista se dice que esta lista es una llamada a una macro. Se produce una expansión de la macro a una función que, acto seguido, es evaluada. Las macros son definibles por el usuario utilizando la función DEFMACRO.

QUOTE y EVAL

A continuación se describen dos importantes expresiones, la forma especial QUOTE y la función EVAL, que controlan la evaluación de las expresiones LISP.

QUOTE es una forma especial que toma un solo argumento y lo retorna sin evaluar, así

(QUOTE DOCENA) → DOCENA

mientras que si se escribiera sólo DOCENA, entonces:

DOCENA → 12

Podría decirse, hablando en términos matemáticos, que es la operación inversa a la evaluación LISP. Es decir, al evaluar la función QUOTE con un argumento, el valor retornado es exactamente el mismo argumento.

Una abreviatura de QUOTE es el apóstrofe que se ha observado antes en algunos ejemplos. (PADRE-DE 'JUAN) es lo mismo que (PADRE-DE (QUOTE JUAN)).

La siguiente secuencia de operaciones ilustra lo dicho:

```
(SETQ JUAN '(SR GARCIA))  ->       JUAN

JUAN   ->          (SR GARCIA)

(PADRE-DE 'JUAN)    ->           PEDRO

(PADRE-DE JUAN)     ->     ¡¡¡ERROR!!!, (SR GARCIA) no es
                           argumento válido.
```

En efecto, la función PADRE-DE intenta evaluar sus argumentos, y al evaluar JUAN se encuentra con un tipo de dato que no puede tratar: la lista (SR GARCIA). Sin embargo, si utilizamos 'JUAN la evaluación se detiene y la función encuentra el argumento correcto, que es el símbolo JUAN.

Ya se ha comentado brevemente el funcionamiento del bucle READ-EVAL-PRINT, que se encarga, entre otras cosas, de evaluar las expresiones que lee. También se ha dicho que QUOTE inhibe el funcionamiento de la parte del bucle que realiza la evaluación; pues bien, la función EVAL proporciona un nivel de evaluación suplementario. EVAL es, por tanto, el inverso de QUOTE; produce una evaluación extra de su argumento. Unos ejemplos ilustrarán esta función:

```
'JUAN  ->      JUAN

(EVAL 'JUAN)   ->      (SR GARCIA)

(SETQ PEDRO '(SR GARCIA SENIOR))  ->  PEDRO

(EVAL (PADRE-DE 'JUAN))   ->      (SR GARCIA SENIOR)
```

El último ejemplo merece una explicación más detallada. EVAL, al ser una función, evalúa su argumento: PADRE-DE 'JUAN. Esta expresión retorna PEDRO y se produce una evaluación más de PEDRO por el efecto propio de EVAL, resultando (SR GARCIA

SENIOR), que se acababa de asignar a PEDRO. Veamos, para finalizar, otra serie de ejemplos:

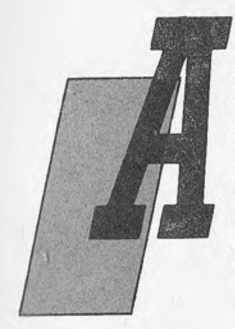
```
(+ 2 2) -> 4
'(+ 2 2) -> (+ 2 2)
(EVAL '(+2 2)) -> 4      Hay una evaluación extra.

'''FOO -> ''FOO
(EVAL '''FOO) -> 'FOO  Se han producido dos evaluaciones
(EVAL (EVAL '''FOO)) -> FOO   Tres evaluaciones
(EVAL (EVAL (EVAL '''FOO))) -> ¡¡¡ERROR!!! FOO, variable
                                no ligada.

(SETQ X '(* 2 3)) -> (* 2 3)
X -> (* 2 3)
(EVAL X) -> 6
(EVAL 'X) -> (*2 3)
'X -> X
```


CAPITULO IV

FUNCIONES DE CONSTRUCCION Y MANEJO DE LISTAS



Aunque no hemos tratado todavía en detalle el comportamiento de las funciones, vamos a introducir en este capítulo las funciones básicas que permiten trabajar con listas en LISP. Se puede, dividir en tres grupos claramente diferenciados: los **constructores** de listas, los **analizadores** y los **predicados** sobre listas.

Constructores

Permiten, como su nombre indica, construir listas partiendo de otros objetos LISP. Si recuerda la definición formal de lista dada en el capítulo 2 observará que, a priori, sólo se dispone de la lista vacía "( )" o NIL, para construir cualquier otro tipo de listas. Partiendo de ella disponemos de la función CONS, abreviatura de CONStructor, para realizar la construcción de otras.

La sintaxis de CONS es la siguiente:

(CONS expresión$_1$ expresión$_2$)

El efecto de CONS, cuando el valor de **expresión$_2$** es una lista, es añadir el valor de **expresión$_1$** a la cabeza de dicha lista. CONS produce una nueva célula elemental cuyo primer puntero apunta al resultado de evaluar **expresión$_1$**. Posteriormente apunta el segundo puntero a la cabeza de **expresión$_2$**, convirtiendo la nueva célula en la primera de la lista. Así, por ejemplo:

```
(CONS 'FOO ( )) -> (FOO)
(CONS 'BAR '(FOO)) -> (BAR FOO)
(SETQ X 'FOO) -> FOO
(SETQ Y 'BAR) -> BAR
(CONS 'X (CONS 'Y ( ))) -> (X Y)
(CONS X (CONS Y ( ))) -> (FOO BAR)
```

(No es necesario que la lista vacía vaya precedida por QUOTE, dado que se evalúa a sí misma.)

Si **expresión$_2$** no es una lista CONS produce un par ordenado, como se observa a continuación:

```
(CONS '(FOO BAR) 'BAZ) ->    ((FOO BAR).BAZ)
(CONS 'A 'B)   ->     (A.B)
```

Dado que CONS es una función, evalúa sus argumentos; así, construcciones como la siguiente son posibles:

```
(SETQ X 'FOO)    ->    FOO
X   ->      FOO
(SETQ X (CONS X '(BAR)))   ->   (FOO BAR)
X    ->      (FOO BAR)


(SETQ X '(A B C)) -> (A B C)
(CONS 'Z X) -> (Z A B C)
X -> (A B C)
```

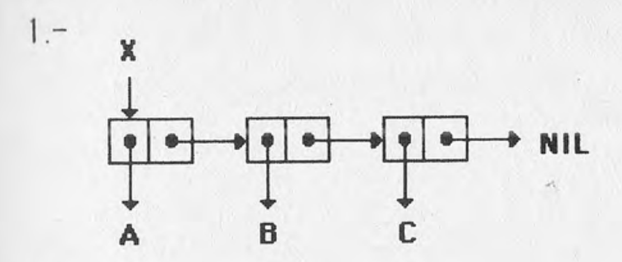
Nótese que, a pesar de haber añadido un nuevo elemento a la cabeza de la lista "X", ésta no ha variado. La representación celular de la figura 1 insiste en estas ideas.

Además de construir listas partiendo de la definición, es también posible construirlas a partir de elementos. Esta es la tarea de LIST y APPEND, cuya sintaxis es:

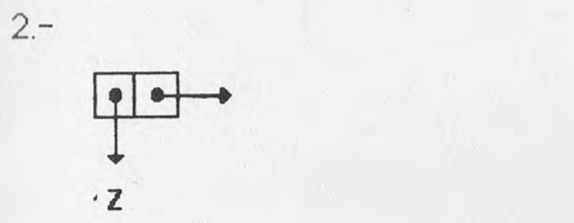
(LIST expresión$_1$ .... expresión$_N$)
(APPEND lista$_1$ .... lista$_N$)

LIST construye una lista utilizando como elementos los resultados de evaluar **expresión$_1$ ... expresión$_N$**:

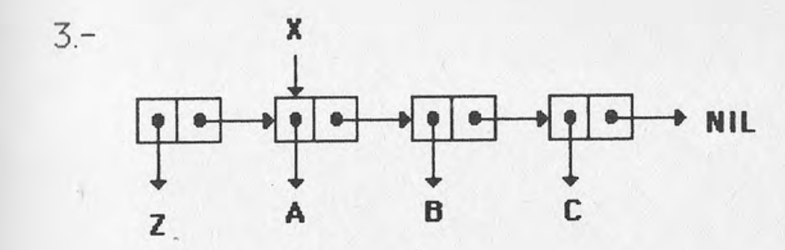
1.-



2.-



3.-



1.- El simbolo X se evalua a la lista (A B C) ligada con él.

2.- Se crea una celula nueva con su CAR dirigido a Z

3.- Se dirige el CDR de la nueva celula a la lista (A B C)

Figura 1.—Proceso de evaluación de (CONS 'Z X)

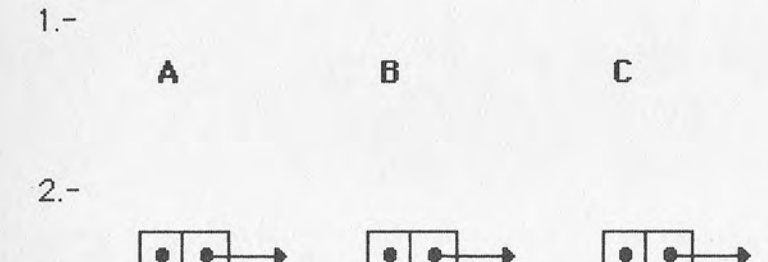
```
(LIST 'A 'B 'C)   ->       (A B C)
(LIST '(FOO BAR) 'BAZ)   ->    ((FOO BAR) BAZ)
(LIST (CONS 'A 'B) 'D)   ->    ((A.B) D)
```

APPEND une todas las listas-argumento en una sola lista.

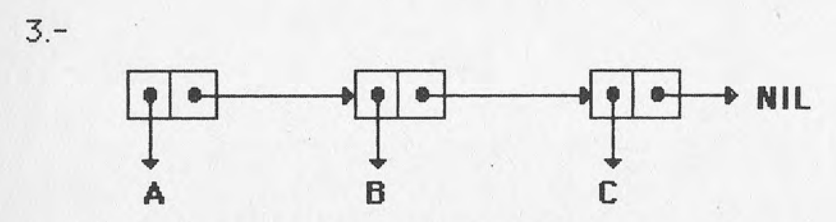
```
(APPEND '(FOO BAR) '(BAZ) '(A B C)) ->
                              (FOO BAR BAZ A B C)

(APPEND (CONS 'A '(FOO)) '(C D)) -> (A FOO C D)
```

Para ilustrar la diferencia entre LIST y APPEND es útil presentar su estructura interna. LIST, como se observa en la figura 2, construye una nueva lista con nuevas células elementales. APPEND (Fig. 3) copia las estructuras de todas las listas, excepto la última,



3.-



1.- Se evaluan los argumentos: A,B Y C

2.- Creación de tantas celulas elementales nuevas como argumentos haya.

3.-Organización de los punteros en forma de lista.

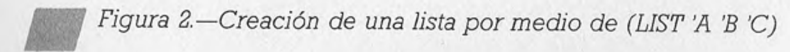
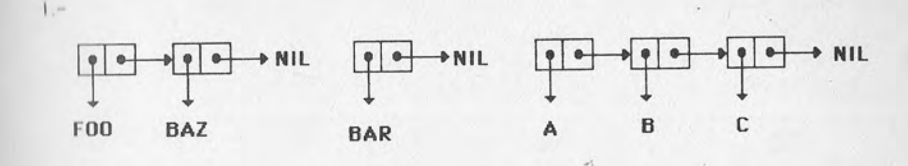
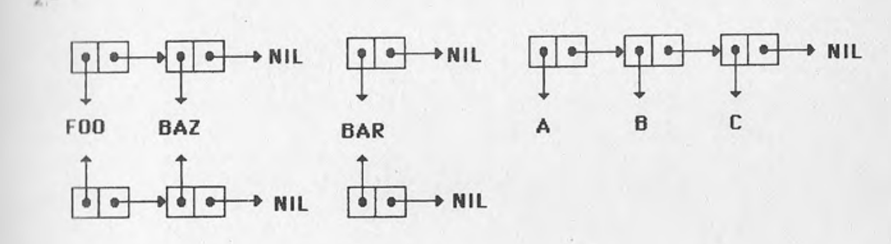


Figura 2.—Creación de una lista por medio de (LIST 'A 'B 'C)

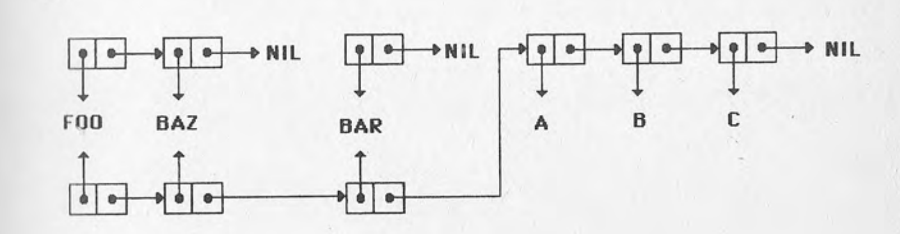
1.-



2.-



3.-



1.- Se evaluan los argumentos.

2.- Se construyen coppias de la estructura de todas las listas excepto la última.

3.- Se reorganizan los punteros para dar la lista resultado con las copias de las primeras y la última.

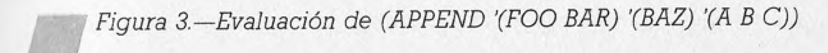


Figura 3.—Evaluación de (APPEND '(FOO BAR) '(BAZ) '(A B C))

y las enlaza entre sí y a la última. De esta forma, por ejemplo, la lista resultante de unir otras dos comparte todas las células de la segunda.

Analizadores de listas

Son funciones que manipulan listas, extrayendo algunos de sus elementos. Las más importantes son **CAR, FIRST, CDR, REST, LAST** y algunas derivadas de éstas.

CAR y FIRST son equivalentes; devuelven el primer *elemento* de la lista que se les proporciona como argumento.

```
(CAR '(FOO BAR)) -> FOO
(FIRST '(FOO BAR)) -> FOO

(SETQ X '(A B C )) -> (A B C)
(CAR X) -> A

(FIRST (FIRST '((A B) C))) -> A
(FIRST '((A B) C)) -> (A B)

(CAR (CAR '(A B C)) -> ¡¡¡ERROR!!! A no es una lista
```

CDR y REST también son equivalentes; retornan la *lista* excluido su primer elemento. Así, por ejemplo:

```
(CDR '(FOO BAR BAZ)) -> (BAR BAZ)
(REST '(FOO BAR BAZ)) -> (BAR BAZ)
(REST (REST '(A B C))) -> (C)
(REST '(A)) -> ( )
(REST 'FOO) -> ¡¡¡ERROR!!! FOO no es una lista

(SETQ FOO '(A B C)) -> (A B C)
(REST FOO) -> (B C)
```

Por supuesto, se pueden combinar todas estas funciones para extraer el elemento o segmento de lista que se desee. Por ejemplo:

```
(CAR (CDR '(1 2 3))) -> 2
(CAR (CDR (CDR '(1 2 3)))) -> 3
(CDR (CAR '(1 2 3))) -> ¡¡¡ERROR!!! 1 no es una lista

(SETQ A '(FOO BAR BAZ)) -> (FOO BAR BAZ)
(CDR (EVAL (CAR '(A B C)))) -> (BAR BAZ)
```


Este último ejemplo es interesante comentarlo con más detalle. Siempre que evalúa una función, LISP evalúa primero sus argumentos. Eso debe hacer con el de CDR. Para ello debe evaluar el argumento de EVAL y, por consiguiente, el de CAR; éste, por efecto del QUOTE, resulta (A B C). De aquí se extrae el CAR que es "A", EVAL actúa sobre "A" y devuelve su contenido: (FOO BAR BAZ). De esta lista se toma el CDR, que resulta (BAR BAZ).

Las combinaciones de CAR y CDR se pueden abreviar en la forma CxxxxR, donde las "x" se pueden sustituir indistintamente por Aes y Des hasta un máximo de cuatro. También existen unas funciones llamadas con los ordinales en inglés: SECOND, THIRD, FOURTH, etc., hasta el décimo (TENTH), que devuelven el correspondiente *elemento* de la lista.

```
(CADR '(A B C)) -> B = (CAR (CDR '(A B C)))
(CADDR '(A B C)) -> C = (CAR (CDR (CDR '(A B C))))
(THIRD '(A B C)) -> C
```

Una función más general es NTH, que tiene la siguiente sintáxis:

(NTH número lista)

Esta función extrae el *elemento* situado en la posición **número** de **lista**.

```
(NTH 2 '(A B C)) -> B
```

La función LAST devuelve la *lista* formada por el último elemento de la lista argumento.

```
(LAST '(A B C)) -> (C)
```

Predicados sobre listas

En LISP existen una serie de funciones lógicas que proporcionan como resultado la certeza o falsedad de alguna característica de los argumentos: Son los **predicados.**

Mientras que para indicar falsedad sólo se puede utilizar NIL, cualquier otra expresión se interpreta como certeza, aunque el símbolo normal es T.

En lo referente a las listas, los principales predicados son **LISTP, ATOM, NULL y ENDP.**

LISTP y ATOM son complementarios. LISTP es cierto si su argumento es una lista y falso en cualquier otro caso. ATOM es cierto cuando es un elemento "atómico" y falso en cualquier otro caso. Su uso puede verse en los siguientes ejemplos:

```
(ATOM 'A) -> T

(SETQ A '(X Y Z)) -> (X Y Z)

(ATOM A) -> NIL

(ATOM '(B C)) -> NIL

(LISTP A) -> T

(LISTP 'A) -> NIL
```

NULL y ENDP detectan ambos la lista vacía o el fin de una lista. NULL es más general, por cuanto admite cualquier tipo de argumento, mientras que ENDP sólo puede tratar listas.

```
(NULL NIL) -> T

(NULL 'A) -> NIL

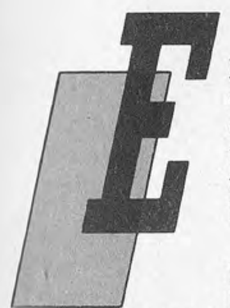
(ENDP 'A) -> ¡¡¡ERROR!!! A no es una lista

(ENDP (CDR '(A))) -> T
```

Con lo dicho hasta aquí, la utilización de listas en un primer nivel no debe plantear problemas. En el capítulo 6 se tratarán nuevas funciones que nos permitirán llevar a cabo otras operaciones con las listas.

CAPITULO V

FUNCIONES Y VARIABLES



El funcionamiento de LISP, como el de otros lenguajes de programación, gira en torno a dos conceptos básicos: funciones y datos, ya apuntados anteriormente. En este capítulo se hará una introducción al manejo de funciones y de variables en LISP.

Una función es un objeto LISP que requiere, para ser evaluado, un conjunto de argumentos previamente evaluados en su totalidad; representa un procedimiento de manipulación de los mismos.

Como se ha comentado en el capítulo 3, la evaluación de una lista depende de su primer elemento. Si éste no es un símbolo que represente una forma especial o una macro, se asume que la lista es una **llamada a una función.** Todos los restantes elementos de la lista son formas que se evalúan y cuyos valores resultantes son suministrados como argumentos a la función, que genera un valor a partir de ellos.

Hay dos formas de indicar qué función va a utilizarse:

— Utilizar un símbolo (nombre) que la identifique.
— Emplear una Expresión Lambda.

Funciones con nombre. Las primitivas

Cabe diferenciar dos grupos de funciones con nombre:

— **Primitivas:** Están ya definidas en la implantación informática del lenguaje. El usuario puede aplicarlas directamente.

— Funciones definidas por el usuario. Este debe especificar el manejo que harán de sus argumentos antes de aplicarlas.

A continuación pueden verse algunos ejemplos de llamadas a funciones primitivas, seguidos por la evaluación de dichas llamadas:

```
(+ 2 3)  -> 5
(* 3 2 4)  -> 24
(/ 6 3)  -> 2
(* (- 8 4) 3)   -> 12
(MAX 2 3 5)  -> 5
(SQRT (MAX (+ 2 3) (* 3 3) (- 4 2)))  -> 3
(CAR '(FOO BAR BAZ))  -> FOO
(CDR '(FOO BAR BAZ))  -> (BAR BAZ)
```

Cada primitiva tiene su propia sintaxis, que precisa el número, tipo y ordenación de sus argumentos. Por ejemplo, la primitiva SQRT (raíz cuadrada) admite un solo argumento, que debe ser de tipo numérico.

A lo largo de los capítulos posteriores se irán introduciendo, según sea necesario, algunas de las primitivas más importantes de LISP, como ya se hizo en el capítulo 4 con las funciones básicas de manejo de listas.

Funciones de usuario con nombre

La herramienta proporcionada por LISP para que el usuario cree sus propias funciones es la macro DEFUN (abreviatura de DEfinir FUNción), cuya sintaxis es la siguiente:

(DEFUN nombre (param$_1$ param$_2$ ... param$_N$) cuerpo-de-la-función)

En donde:

- nombre: símbolo mediante el cual se identifica la función.
- (param$_1$ param$_2$ ... param$_N$): lista de parámetros. Sus elementos son símbolos que pueden aparecer en el cuerpo de la función y cuyos valores se corresponden con los de los argumentos con los que se llama a la función.

- cuerpo-de-la-función: descripción del procedimiento de manipulación de los parámetros.

DEFUN no evalúa sus argumentos. Sólo define un nuevo procedimiento a partir de ellos, identificado mediante **nombre.** El valor que retorna DEFUN es precisamente **nombre.**

Para llamar a una función de usuario se utiliza la misma estructura que en el caso de las primitivas: el nombre seguido de los argumentos;

(nombre arg$_1$ arg$_2$ ... arg$_N$)

Un ejemplo ilustrará el empleo de DEFUN y la llamada a una función definida con él:

```
(DEFUN SUMAR (X Z)
  (LIST 'LA 'SUMA 'DE X 'Y 'DE Z 'ES (+ X Z))) -> SUMAR
(SUMAR 3 4) -> (LA SUMA DE 3 Y DE 4 ES 7)
(SUMAR 3 'FOO) -> ¡¡¡error!!!
(SUMAR (+ 3 2) 4) -> (LA SUMA DE 5 Y DE 4 ES 9)
```

Una vez evaluados los argumentos (de SUMAR en este caso) se establece una correspondencia entre los parámetros y los resultados de dicha evaluación. Cada uno de los parámetros se liga temporalmente al valor del respectivo argumento. Tras esto se lleva a cabo la operación descrita en el cuerpo de la función, cuyo resultado es el valor retornado por la llamada a la función.

En el cuerpo de la función puede haber varias operaciones. En este caso se evalúan secuencialmente, siendo el valor retornado por el conjunto el resultado de la última.

```
(DEFUN SEC (X Y)
    (* X Y)
    (+ X Y)
    (/ X Y)
    (- X Y) )  -> SEC
(SEC 4 8) -> -4
```

Algunas de estas operaciones pueden ser, incluso, llamadas a otras funciones.

Funciones sin nombre: expresiones Lambda

Las expresiones Lambda permiten definir una función en el momento en que se la utiliza. Una expresión Lambda sustituye a **nombre** en una llamada a función. Su estructura coincide con la de DEFUN, pero sustituyendo:

DEFUN nombre

por

LAMBDA

En el siguiente ejemplo puede verse una misma función definida por los dos métodos.

```
(DEFUN SEGUNDO (L)
        (CAR (CDR L))) -> SEGUNDO

(SEGUNDO '(A B C)) -> B

((LAMBDA (L) (CAR (CDR L))) '(A B C)) -> B
```

Una función definida mediante una estructura Lambda ha de describirse cada vez que se utiliza, por lo que es aconsejable hacer uso de DEFUN si la función es llamada repetidas veces. Posteriormente se presentarán ejemplos que aclararán la verdadera utilidad de las estructuras Lambda.

Variables

Ya se han tratado brevemente los símbolos como una de las estructuras básicas de datos de LISP. Los símbolos pueden representar variables u otras entidades (funciones, canales de entrada/salida, etc.), como en otros lenguajes de programación.

Dado que una de las principales funciones de los símbolos es dar nombre a las variables se emplearán indistintamente ambos términos cuando el contexto sea el adecuado.

En LISP una variable puede almacenar datos: números, símbolos, cadenas de caracteres, etc. Las principales operaciones que se pueden hacer con variables son la recuperación de su contenido y la modificación del mismo. Ya se ha tratado la recuperación del contenido de una variable mediante la evaluación del símbolo que la identifica. A continuación se verá cómo modificar su contenido.

El almacenamiento de datos se realiza físicamente en una porción de memoria que, en terminología LISP, se dice que está **ligada** a la variable. Una variable puede estar ligada a varias posiciones de memoria o, dicho de otro modo, puede tener distintos valores; según el contexto en que se haga referencia a la variable se accederá a uno u otro de ellos.

Para dar un valor a un símbolo se utiliza la forma especial SETQ, que almacena el valor en la porción de memoria ligada a la variable. Se dice que realiza una **asignación.** La sintaxis de SETQ es la siguiente:

(SETQ nom-símbolo$_I$ forma$_I$ nom-símbolo$_2$ forma$_2$ ... nom-símbolo$_N$ forma$_N$)

Donde:

- **nom-símbolo$_I$** es el símbolo que identificará a la variable.
- **forma$_I$** es la expresión que, *una vez evaluada,* será el valor de la variable.

Puesto que SETQ es una forma especial, tiene sus propias características. Estas consisten en que deja sin evaluar los **nom-símbolo$_1$** y evalúa sólo las **forma$_1$**. Comienza por **forma$_1$** y el valor resultante lo **asigna a nom-símbolo$_1$**; de manera análoga procede, secuencialmente, con **forma$_2$**, **nom-símbolo$_2$** y las parejas restantes.

En una forma pueden utilizarse las variables anteriores, con los valores previamente asignados. Lógicamente, el número de formas ha de ser igual al de símbolos.

SETQ retorna el valor de la última forma de la lista.

```
(SETQ X 4) -> 4
X -> 4
'X -> X

(SETQ X 5 Y 'FOO Z '(A B C)) -> (A B C)
X -> 5
Y -> FOO
Z -> (A B C)
```

```
(SETQ F X) -> 5
F -> 5

(SETQ F 'X) -> X
F -> X

(SETQ X (+ 2 3)) -> 5
X -> 5

(SETQ X (A B C)) -> ¡¡¡ERROR!!!
```

Función "A" no definida. El intérprete ha tratado de evaluar la lista (A B C) y ha encontrado que "A" no es un nombre de forma especial, de macro, ni de función.

```
(* A B) -> ¡¡¡ERROR!!!
```

Variable "A" no ligada. Trata de hallar el valor de la variable "A". Como "A" no está ligada a ninguna porción de memoria, no está definida.

```
(SETQ A 2) -> 2
(SETQ B 3) -> 3
(* A B) -> 6
```

SETQ realiza una **asignación** de valor a las variables. En terminología LISP esto quiere decir que actualiza el contenido de la porción de memoria ligada a la variable en el instante en que se evalúa SETQ. Esta puntualización es importante porque ya se ha dicho que una variable puede tener más de una ligadura.

Establecimiento de ligaduras

Hemos visto ya cómo recuperar el valor de una variable y cómo asignárselo. Seguidamente se tratará del establecimiento de las distintas ligaduras y la regla general para establecer qué ligadura es efectiva en cada llamada a la variable.

Con lo estudiado hasta ahora puede decirse que aparece una nueva ligadura cuando se llama a una función. Las variables de

la lista de parámetros se ligan a posiciones de memoria a las que se asignan los valores de los argumentos con los cuales se llamó a la función. A partir de ese instante los contenidos de las variables de la lista de parámetros serán los especificados por las nuevas ligaduras. Cuando termine la evaluación, los valores de las variables volverán a quedar determinados por las ligaduras anteriores a la llamada a la función. Unos ejemplos aclararán estas ideas:

```
(DEFUN CUADRADO (N) (* N N)) -> CUADRADO
N -> ¡¡¡ERROR!!! Variable no ligada.
(CUADRADO 6) -> 36
```

Durante la evaluación N ha quedado ligada a 6. (* N N) → ((* 6 6) → 36 N → ¡¡¡ERROR!!! Variable no ligada.

```
(SETQ X 3)  -> 3
(SETQ Y 5)  -> 5
(DEFUN F (X Y)
      (+ (* X Y) 10))  -> F
X -> 3
Y -> 5
(F 1 2) -> 12
```

Durante el período de evaluación de F las variables "X" e "Y" se ligan, de modo temporal, a 1 y 2, respectivamente.

```
X -> 3
Y -> 5
```

Las variables recuperan sus valores iniciales.

```
(DEFUN DOBLE (X) (+ X X)) -> DOBLE
(DEFUN CUADRUPLE (X) (DOBLE (DOBLE X))) -> CUADRUPLE
X -> ¡¡¡ERROR!!! Variable no ligada
(CUADRUPLE 10)              X ligada a 10 en CUADRUPLE.
  (DOBLE (DOBLE 10))        X ligada a 10 en DOBLE.
                            se enmascara la ligadura
                            anterior.
```

```
(DOBLE 10)
     (+ 10 10) -> 20          Desaparece la ligadura
                              de DOBLE.
(DOBLE 20)                    X ligada a 20 en DOBLE.
                              Vuelve a quedar enmascarada
                              la ligadura de CUADRUPLE.
     (+ 20 20) -> 40          Desaparece la ligadura
                              de DOBLE.
(CUADRUPLE 10) -> 40          Desaparece la ligadura de
                              CUADRUPLE.
X -> ¡¡¡ERROR!!! Variable no ligada
```

En la figura 1 vuelve a encontrarse este ejemplo, ilustrado de otro modo.

Las ligaduras creadas en la llamada a una función se denominan **ligaduras locales.** Permanecen vigentes durante la evaluación de la función y desaparecen cuando termina ésta.

Cuando se llama a una variable se recupera el contenido de la posición de memoria ligada a ella en ese instante. Si no se ha realizado ninguna ligadura la variable no tiene valor definido, de ahí los errores mostrados antes.

Ya se ha comentado el funcionamiento de SETQ, que asigna un valor a la ligadura vigente en ese instante. Teniendo esto en cuenta es fácil ver que, al terminar la evaluación de una función y desaparecer las ligaduras asociadas a ella, se pierden los valores asignados en el cuerpo de la función.

```
(DEFUN CAMBIO (X)
     (PRINT X)
     (SETQ X '(FOO BAR BAZ)))
(SETQ LISTA '(A B C)) -> (A B C)
(CAMBIO LISTA) ->   (A B C)
                    (FOO BAR BAZ)
LISTA -> (A B C)
X -> ¡¡¡ERROR!!! Variable no ligada.
```

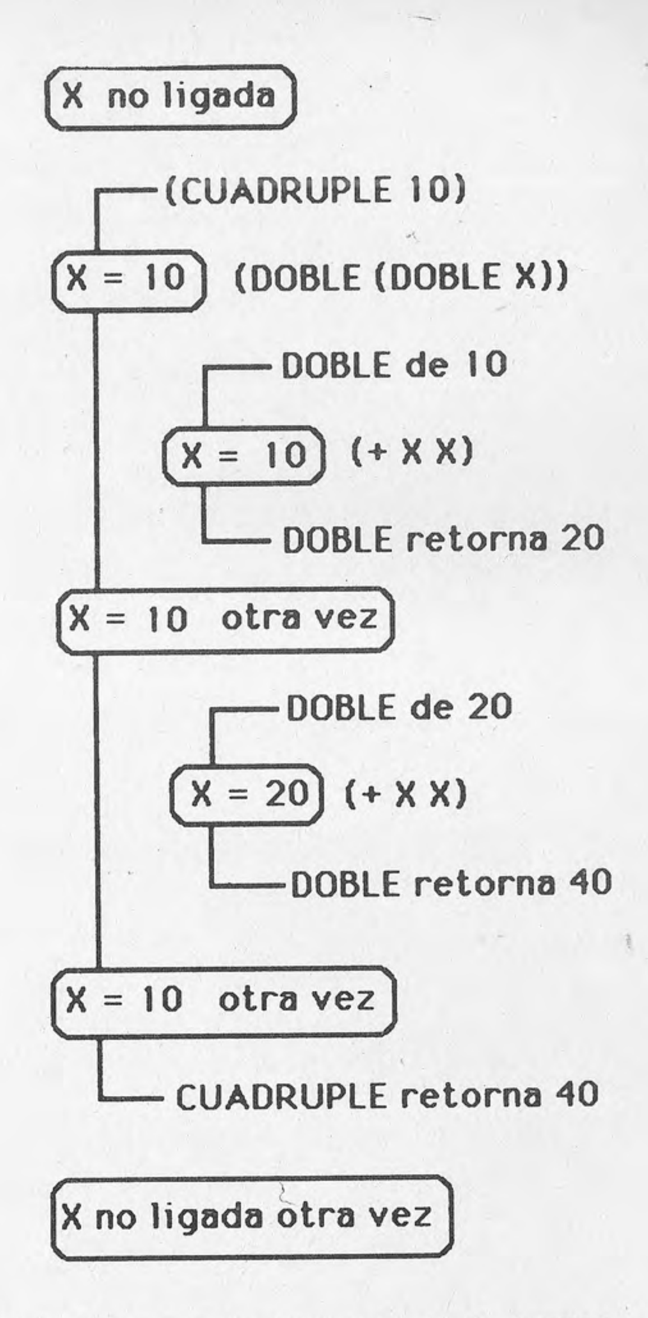



Figura 1.—Ligaduras locales en la función CUADRUPLE.

Se llama a la función CAMBIO con el argumento LISTA. El valor retornado por la evaluación de éste, (A B C), se liga a "X". La función PRINT, que se tratará en un capítulo posterior, muestra en pantalla el valor de "X". SETQ asigna un nuevo contenido, (FOO BAR BAZ), a la ligadura vigente de "X". Esta lista se retorna como resultado de la evaluación de la función. Una vez terminado el proceso, la ligadura de "X" desaparece.

Si se utiliza SETQ con un símbolo que no está afectado de ninguna ligadura previa y que, por tanto, no tiene ningún valor que actualizar, se produce una **ligadura global**. Este nombre define una ligadura cuyo valor asociado es accesible desde cualquier punto del programa, siempre que no esté enmascarado por una ligadura local. En este caso SETQ no se limita a modificar el contenido asociado a una ligadura, sino que la establece.

```
(SETQ X 2) -> 2

(DEFUN DUPLICAR (Y)

     (* X Y))        -> DUPLICAR

(DEFUN TRIPLICAR (X)

     (PRINT X)

     (*3 X))         -> TRIPLICAR

(DUPLICAR 4) -> 8
```

El valor de "X", ligado de forma global, puede ser llamado desde el cuerpo de la función.

```
(TRIPLICAR 5) -> 5

                 15
```

Durante la evaluación de triplicar, la ligadura local de "X" a 5 enmascara la ligadura global a 2.

```
X -> 2
```

Vuelve a ser efectiva la ligadura global.

```
(DEFUN LIGAR ()

     (SETQ FOO '(A B C))) -> LIGAR

(LIGAR) -> (A B C)
```


FOO no tenía ninguna ligadura previa. Así pues, este SETQ realiza una ligadura global.

```
FOO -> (A B C)
```

En Common LISP los efectos de las ligaduras locales, a diferencia de las globales, son internos al cuerpo de las expresiones que las crean y no se transmiten a funciones llamadas desde éste. Véase un ejemplo que utiliza la función DUPLICAR con una nueva definición.

```
(SETQ X 2) -> 2

(DEFUN DUPLICAR ()

     (* X Y)) -> DUPLICAR

(DEFUN CUADRUPLICAR (Y)

     (* 2 (DUPLICAR))) -> CUADRUPLICAR

(CUADRUPLICAR 3) -> ¡¡¡ERROR!!!
```

La variable "Y" utilizada en el cuerpo de DUPLICAR no está ligada, ya que la ligadura local establecida por CUADRUPLICAR no alcanza a DUPLICAR. Redefinimos ambas funciones de forma correcta.

```
(DEFUN DUPLICAR (Y)

     (* X Y)) -> DUPLICAR

(DEFUN CUADRUPLICAR (Y)

     (* 2 (DUPLICAR Y))) -> CUADRUPLICAR

(CUADRUPLICAR 3) -> 6
```

La variable "Y" aparece como argumento y parámetro de DUPLICAR. Al llamar a CUADRUPLICAR aparece una primera ligadura entre "Y" y 3. Cuando se llama a DUPLICAR desde el cuerpo de CUADRUPLICAR "Y" vale 3. Esto determina la *nueva* ligadura que aparece en DUPLICAR, que *también* une "Y" y 3.

Ya que en la primera definición de DUPLICAR no había lista de parámetros, todas las variables referidas en el cuerpo deben buscar sus valores en las ligaduras globales. En el caso de "X" ésta existe, pero con "Y" no sucede así.

Los nombres de las funciones también son símbolos, por lo que todas estas reglas les son también aplicables.

Puede decirse, en cierto modo, que DEFUN realiza con las fun-

ciones una tarea semejante a la que SETQ lleva a cabo con las variables, permitiendo asignar globalmente una función a un nombre. Normalmente, todos los nombres de función que se manejan son globales. En particular, si se definen dos funciones representadas por el mismo símbolo, la segunda anula a la primera, como ocurría en el ejemplo anterior.

Predicados sobre ligaduras

BOUNDP da como resultado T si se aplica a una variable ligada; en otro caso es NIL. Su sintaxis es muy sencilla:

(BOUNDP nombre-símbolo)

FBOUNDP retorna T si el símbolo es el nombre de una función, una macro o una forma especial, y es falso en caso contrario.

(FBOUNDP nombre-símbolo)

MAKUNDBOUND elimina la ligadura vigente de una variable.

(MAKUNDBOUND nombre-símbolo)

FMAKUNBOUND hace lo mismo con funciones. Ambos retornan **nombre-símbolo.**

(FMAKUNBOUND nombre-símbolo)

Veamos algunos ejemplos:

```
(BOUNDP 'X) -> NIL
(SETQ X 'FOO) -> FOO
(BOUNDP 'X) -> T
(MAKUNBOUND 'X) -> X
```

Nótese la presencia del apóstrofe, ya que el argumento es el nombre del símbolo, y no la variable representada por éste.

```
X -> ¡¡¡ERROR!!! Variable no ligada.
(DEFUN FOO (X)
     (* X 2)) -> FOO
(FOO 3) -> 6
```



```
(FBOUNDP 'FOO) -> T
(FMAKUNBOUND 'FOO) -> FOO
(FOO 3) -> ¡¡¡ERROR!!! Función FOO no definida.
```

La función APPLY

Ya hemos comentado que cualquier lista que se va a evaluar y que no puede interpretarse como una llamada a una macro o una forma especial, se considera una llamada a una función. APPLY es un modo especial de llamar funciones. Su sintaxis es, de forma simplificada, la siguiente:

(APPLY función lista)

APPLY llama a **función** utilizando como argumentos los elementos de **lista,** que tienen que coincidir en número y tipo con los argumentos que admita **función.**

```
(APPLY #'+ '(1 2)) -> 3    = (+ 1 2)
(SETQ X '(3 5)) -> (3 5)
(APPLY #'+ X) -> 8
(SETQ SUMA #'+) -> +
(APPLY SUMA X) -> 8
```

Nótese que en los ejemplos las funciones van precedidas de #'. Esta indicación es la abreviatura de la forma especial FUNCTION, que tiene un efecto semejante al de QUOTE sobre símbolos ligados a funciones y estructuras LAMBDA. En general, puede sustituirse por QUOTE, pero resulta conveniente emplear #' por motivos que van más allá del propósito de esta obra y que no se detallarán.

Efectos laterales

Al evaluar cualquier expresión LISP se produce un resultado que aparece en pantalla. Además de esto, que se puede considerar el efecto principal, pueden producirse **efectos laterales,** entendiendo por tales otros procesos que no tienen por qué ser visibles en la pantalla.

Hay formas cuya utilidad primaria es el efecto principal, es de-

cir, el resultado de la evaluación. Entre ellas, los predicados como LISTP, BOUNDP, NULL, etc., las primitivas aritméticas o las funciones básicas de manejo de listas, como CAR, CDR, LIST, etc.

Hay otras formas cuyo interés primario se basa en su efecto lateral. Dos ejemplos claros son la macro DEFUN y la forma especial SETQ. DEFUN liga una función a un símbolo, además de retornar el nombre del mismo, cosa que en general carece de interés. SETQ asigna un valor a una variable y lo devuelve como resultado. Otras muestras de la importancia de los efectos laterales son las estructuras de control, que se tratarán posteriormente.

La macro SETF

SETQ es la forma más sencilla de asignación de variables. SETF es la forma general de asignación de valor a una estructura de datos LISP. Las sintaxis de ambas son semejantes:

(SETF $lugar_1$ $forma_1$ $lugar_2$ $forma_2$ ... $lugar_N$ $forma_N$)

Donde **$lugar_I$** es una forma que señala una determinada estructura de datos. A diferencia de lo que ocurría con SETQ, en algunos casos se evalúa. Puede ser, entre otras:

- Un nombre de variable.
- Una llamada a una función de acceso a un elemento de una lista, como CAR, CDR, THIRD, etc.
- Una función que accede a un elemento de una estructura (véase capítulo 12).

SETF se expande a la expresión adecuada para realizar la asignación del valor resultante de la evaluación de **$forma_I$** a la estructura indicada por **$lugar_I$**. En el caso particular de que **$forma_I$** sea un nombre de variable, SEFT se expande a SETQ. Con el empleo de SETF se puede prescindir del uso de SETQ. El motivo de que se mantenga esta forma especial en Common LISP es su importancia histórica en el desarrollo del lenguaje.

```
(SETF X '(FOO BAR BAZ)) -> (FOO BAR BAZ)  =
                            (SETQ X '(FOO BAR BAZ))

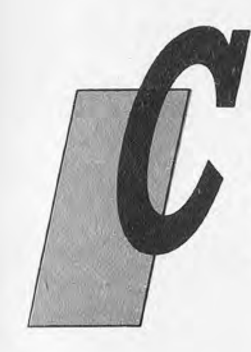
(SETF (CAR X) 'A) -> A    = (RPLACA X 'A)
```

(Véase Cap. 6)

```
X -> (A BAR BAZ)
```


CAPITULO VI

MANEJO AVANZADO DE LISTAS



Como hemos podido comprobar a lo largo de esta obra, las listas son un tipo de datos muy potente y de gran versatilidad. En este capítulo se van a presentar, aunque no de forma exhaustiva, un segundo grupo de funciones que permiten utilizar eficientemente estas construcciones.

Predicados de igualdad

Como paso preliminar vamos a tratar las formas en que LISP puede entender la igualdad entre dos objetos. Se distinguen cuatro niveles de igualdad entre objetos, por lo que existen cuatro predicados distintos que son, ordenados del más al menos restrictivo: EQ, EQL, EQUAL y EQUALP.

Si dos objetos LISP hacen cierto el predicado EQ son físicamente iguales, es decir, son el mismo; ocupan la misma posición *física* de memoria. Así pues, si se hace:

```
(SETQ X 'A) -> A

(SETQ Y X) -> A
```

obtendremos:

```
(EQ X Y) -> T
```

sin embargo, si ahora se introduce:

```
(SETQ Y '(A)) -> (A)
(SETQ X (CONS 'A NIL)) -> (A)
```

resultará:

```
(EQ X Y) -> NIL
```

pues "X" e "Y" representan objetos en diferentes posiciones de memoria. Incluso con los números puede suceder (según la implantación) que, por ejemplo:

```
(EQ 3 3) -> NIL
```

y siempre sucede que, por ejemplo:

```
(EQ 3 3.0) -> NIL
```

pues no son el mismo, ya que uno es un entero y el otro es un decimal.

EQL es un predicado menos restrictivo. Se comporta como EQ salvo en lo referente a valores numéricos, para los cuales utiliza el concepto matemático de igualdad si son del mismo tipo, así:

```
(EQL 3 3) -> T
(EQL 3 3.0) -> NIL
```

EQUAL considera que dos objetos son iguales si tienen la misma estructura y forma (aunque distingue entre mayúsculas y minúsculas), independientemente de su posición de memoria, salvo en el caso de vectores y matrices, para los que utiliza una comparación tipo EQ. Los ejemplos aclararán cómo funciona.

```
(EQUAL 'A 'B) -> NIL
(EQUAL 'A 'A) -> T
(EQUAL 3 3.0) -> NIL (tienen distinta estructura,
                     3 es entero y 3.0 es decimal)
(EQUAL '(A.B) (CONS 'A 'B)) -> T
(EQUAL 'FOO 'FOO) -> T
(EQUAL 'FOO 'foo) -> NIL
```


El último y más laxo es EQUALP, que relaja las restricciones de las matrices, las mayúsculas y minúsculas, y los tipos de números, así:

```
(EQUALP 3 3.0) -> T
(EQUALP 'FOO 'foo) -> T
```

Para comprender la utilidad y el funcionamiento de estos predicados ha de tenerse en cuenta la diferencia que existe en LISP entre el objeto, su representación y la posición de memoria que ocupa. Puede decirse que dos símbolos iguales ocupan la misma posición de memoria, es decir, son EQ y, por lo tanto, también EQUAL y EQUALP.

Para los números no hay ninguna regla fija, pues dos números iguales pueden o no ocupar la misma posición de memoria según la implantación informática del lenguaje. El predicado EQL y también "=" sirven para determinar la igualdad de números. Dos números iguales pueden no ser EQ, pero si son del mismo tipo sí serán EQUAL.

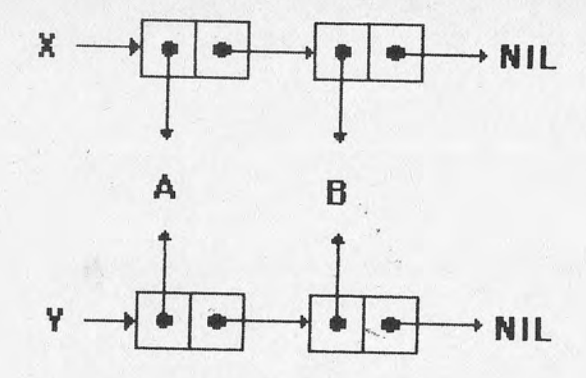
Una lista se identifica por la posición de su primera célula en la memoria. Dos listas que contienen los mismos elementos pueden tener distintas células, aunque los punteros CAR apunten a las mismas posiciones de memoria en ambos casos. Dicho de otro modo, no serán EQ pero sí EQUAL. La figura 1 ilustra esta idea.

Operadores destructivos para manejo de listas

Teniendo presentes estas ideas, es posible ahora introducir en el trabajo con listas la noción de **operador quirúrgico o destructivo,** como contraposición a los vistos en el capítulo 4. Los más importantes son RPLACA, RPLACD, NCONC, PUSH y POP.

Puede decirse que los operadores normales (los **no destructivos)** no modifican la estructura de su argumento, sino que crean una copia de éste. Los **destructivos,** por el contrario, modifican su argumento. Para ilustrar esta diferencia puede compararse APPEND con su equivalente destructivo NCONC:

```
(SETQ FOO '(A B)) -> (A B)
(SETQ BAR '(V X)) -> (V X)
(APPEND FOO '(C D)) -> (A B C D)
(NCONC BAR '(Y Z)) -> (V X Y Z)
```



X --> (A B)
Y --> (A B)
(EQ X Y) --> NIL
(EQUAL X Y) --> T

Figura 1.—Dos listas con los mismos elementos y distintas direcciones de memoria, es decir, distintas células.

Aparentemente el resultado es el mismo, peo si ahora solicitamos:

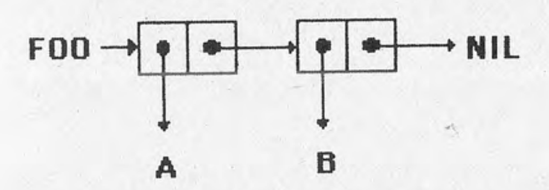
```
FOO -> (A B)
BAR -> (V X Y Z)
```

Es decir, NCONC modifica la estructura de la lista original, mientras que APPEND la reproduce en una nueva lista. NCONC cambia físicamente la última célula de su primer argumento para que su puntero CDR apunte a la primera célula del segundo argumento. La figura 2 muestra la secuencia que sigue APPEND y la figura 3 la que sigue NCONC.

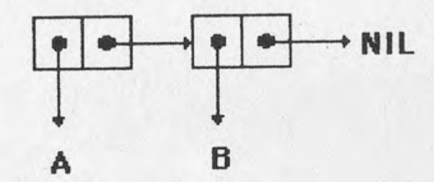
RPLACA tiene el efecto de sustituir el CAR de una lista por lo que se le indique. Su sintaxis es

(RPLACA lista objeto)

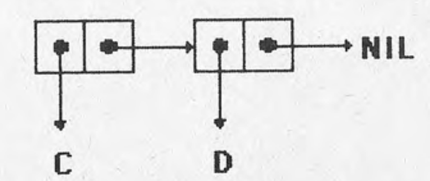
La lista original aparece como:



Al ejecutar Append,se crea una copia de ésta:



y una nueva lista, para almacenar (C D)



Despues une estas dos últimas:

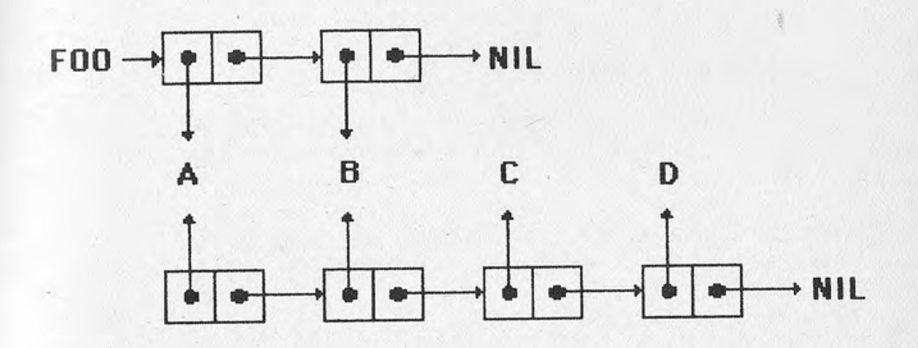
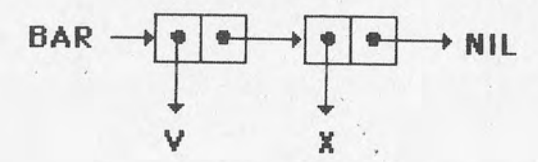
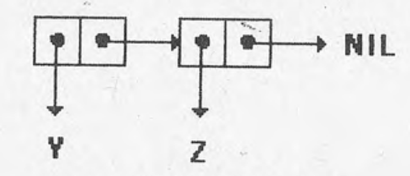


Figura 2.—Evaluación de (APPEND FOO '(C D)), estando FOO ligado a (A B).

La lista original aparece como:



Se crea una nueva lista:



y se unen las dos.

BAR → NIL

Y Z Y Z

La lista resultante completa queda ligada a BAR


Figura 3.—(NCONC BAR '(Y Z), estando BAR ligado a (V X).

y actúa reemplazando el CAR de **lista** por **objeto.** Su equivalente no destructivo sería:

(CONS objeto (CDR lista))

que, al ser evaluado, retorna una lista con las mismas características que la retornada por RPLACA y no modifica **lista.** Para obtener exactamente el mismo efecto se puede utilizar:

(SETQ lista (CONS objeto (CDR lista)))

Junto con RPLACA conviene tratar RPLACD, que es análogo pero reemplaza el CDR de la lista. Su sintaxis y equivalente no destructivo serían, respectivamente:

(RPLACD lista objeto)
(LIST (CAR lista) objeto)

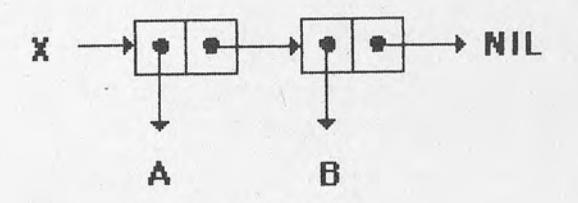
Algunos ejemplos del uso de RPLACA y RPLACD son:

```
(SETQ X '(A C)) -> (A C)
(RPLACA X 'B) -> (B C)
X -> (B C)
(RPLACA X '(A B)) -> ((A B) C)
X -> ((A B) C)
(RPLACD X '(C D)) -> ((A B) (C D))
X -> ((A B) (C D))
```

Por supuesto, la utilización de estas funciones es muy poderosa, pero encierra riesgos importantes. A continuación se dan ciertas estructuras que se pueden conseguir, entre muchas otras, con sus representaciones celulares. Se supondrá en cada operación que "X" está ligada a (A B).

```
(RPLACD X X) -> (A A A A A ....) (Fig. 4)
(RPLACA X X) -> (((((( ..... (Fig. 5)
(SETQ Y '((A B) (C D))) -> ((A B) (C D))
```

La lista



se transforma en

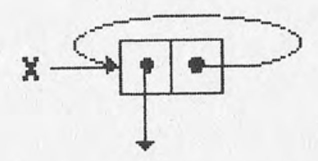


Figura 4.—Creación de una lista circular mediante (RPLAC X X), siendo (A B) el valor de X.

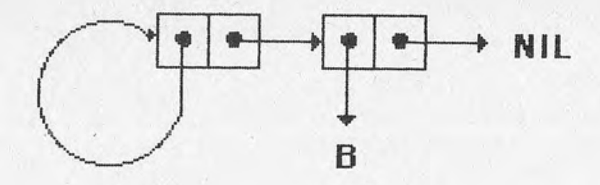
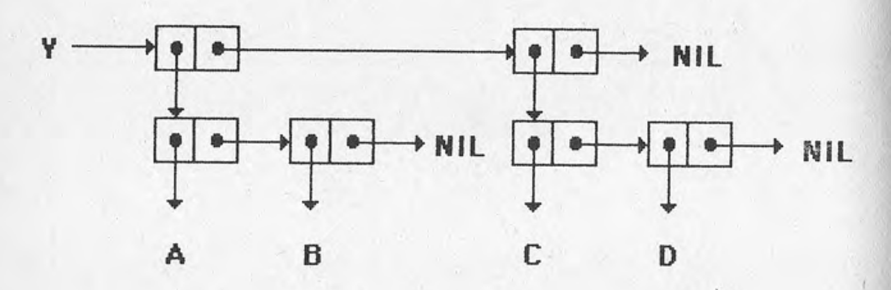
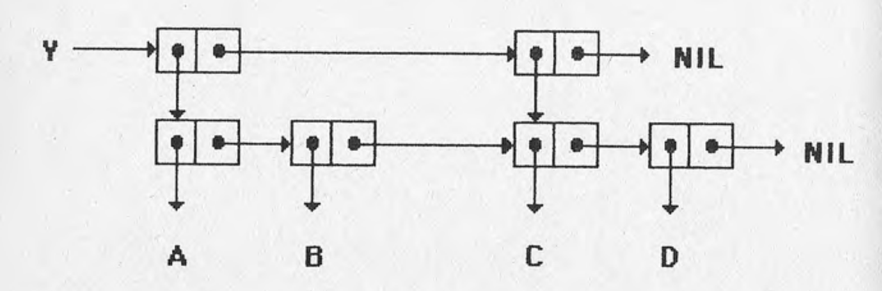


Figura 5.—Resultado de la evaluación (RPLACA X X), estando X ligado a (A B).

La lista original es:



Y el resultado:



Como se puede observar, NCONC actua sobre las listas: (FIRST Y) y (SECOND Y)

Figura 6.—Resultado de (NCONC (FIRST Y) (SECOND Y)), siendo ((A B (C D)) el valor de Y). Se obtiene ((A B C D) (C D).


```
(NCONC (FIRST Y) (SECOND Y)) -> (A B C D)

Y -> ((A B C D) (C D))  (Fig. 6)

(SETQ Y '(C D)) -> (C D)

(NCONC X Y) -> (A B C D)

(NCONC X Y) -> (A B C D C D C D ... (Fig. 7)

Y -> (C D C D C D C D ...
```

PUSH y POP son macros que tratan a la lista como si fuera una "pila" de objetos LISP.

(PUSH objeto lista)

PUSH introduce **objeto** en la cabeza de **lista**, como haría CONS, pero de forma destructiva, cambiando el valor de **lista.** PUSH evalúa el nuevo valor de lista.

```
(SETQ X '(BAR BAZ)) -> (BAR BAZ)

(CONS 'FOO X) -> (FOO BAR BAZ)

X -> (BAR BAZ)

(PUSH 'FOO X) -> (FOO BAR BAZ)

X -> (FOO BAR BAZ)
```

La macro POP es complementaria de PUSH: extrae el primer elemento de la lista. En cierto modo es equivalente a un CAR quirúrgico. Siguiendo con los ejemplos anteriores tendríamos:

```
(CAR X) -> FOO

X -> (FOO BAR BAZ)

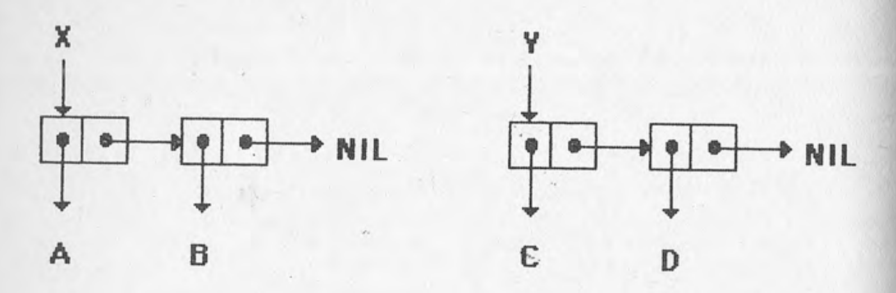
(POP X) -> FOO

X -> (BAR BAZ)
```

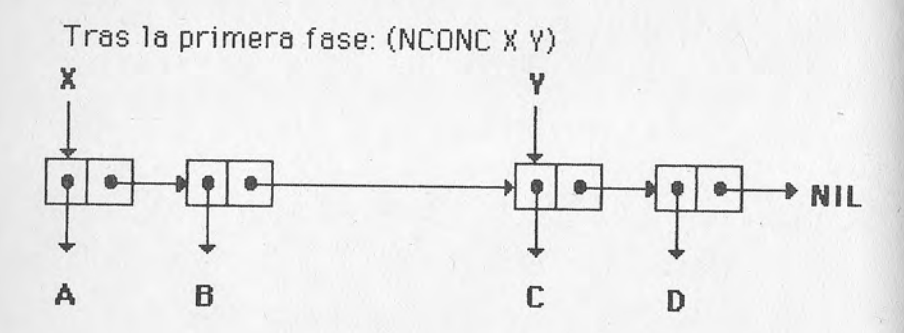
Otros dos grupos de funciones relacionadas, uno destructivo y el otro no, pero de parecidos resultados, son las familias REMOVE y DELETE:

REMOVE	DELETE
REMOVE-IF	DELETE-IF
REMOVE-IF-NOT	DELETE-IF-NOT
REMOVE-DUPLICATES	DELETE-DUPLICATES

Las listas originales son:



Tras la primera fase: (NCONC X Y)



Segunda fase : (NCONC (NCONC X Y) Y)

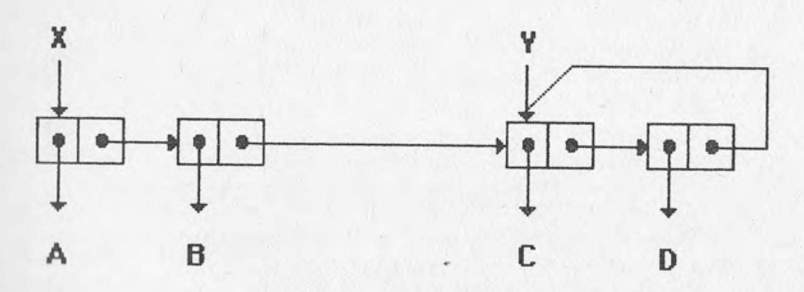


Figura 7.—Aplicación sucesiva de la función NCONC.

La familia REMOVE copia sus argumentos y la familia DELETE los altera.

(REMOVE objeto lista)
(DELETE objeto lista)

Ambos eliminan los elementos de **lista** que sean iguales en sentido EQL a **objeto.**

(REMOVE-IF predicado lista)
(DELETE-IF predicado lista)

eliminan los elementos de **lista** que cumplen **predicado.** Análogamente, REMOVE-IF-NOT y DELETE-IF-NOT eliminan los elementos que no satisfacen **predicado.**

(REMOVE-DUPLICATES lista)
(DELETE-DUPLICATES lista)

eliminan todos los elementos repetidos en **lista** salvo la primera ocurrencia de cada clase.

```
(SETQ X '(A B C A B C)) -> (A B C A B C)
(REMOVE-DUPLICATES X) -> (A B C)
X -> (A B C A B C)
(DELETE-DUPLICATES X) -> (A B C)
X -> (A B C)
(SETQ Y '(A 2 3 FOO)) -> (A 2 3 FOO)
(REMOVE-IF 'NUMBERP Y) -> (A FOO)
Y -> (A 2 3 FOO)
(DELETE-IF 'NUMBERP Y) -> (A FOO)
Y -> (A FOO)
```

Normalmente todas estas funciones utilizan como predicado de igualdad EQL, aunque suele ser posible especificar otra función para las comparaciones.

Las listas como conjuntos y como tablas

Hay otras formas de considerar las listas además de las vistas. Aquí se expondrán sólo las dos más significativas: como conjuntos y como tablas (llamadas también *listas de asociación).* El tratamiento de conjuntos se realiza, entre otras, con las funciones:

UNION, INTERSECTION, MEMBER, MEMBER-IF
y SET-DIFFERENCE

que tienen sus versiones quirúrgicas en

NUNION, NINTERSECTION y NSET-DIFFERENCE

aquí sólo se expondrán las no destructivas, entendiéndose que éstas últimas son análogas. También se supondrá en adelante en este capítulo que la igualdad se considera en sentido EQL.

(UNION lista$_1$ lista$_2$)

produce la unión de las listas dejando, si hay elementos iguales, sólo un elemento de cada clase. En general no respeta la ordenación de las listas originales.

(INTERSECTION lista$_1$ lista$_2$)

produce una copia de **lista$_1$** en la que se han quitado todos los elementos que no son iguales a alguno de **lista$_2$** con los mismos criterios que las dos funciones anteriores.

(MEMBER objeto lista)

comprueba si **objeto** es igual a algún elemento de **lista;** si lo es, devuelve el resto de la misma lista a partir del elemento. Adviértase que *MEMBER devuelve una parte de la misma lista, no una copia.* Si **objeto** no es miembro de **lista,** retorna NIL.

```
(MEMBER '2 '(1 2 3) -> (2 3)
```

(MEMBER-IF predicado lista)

trabaja análogamente a MEMBER, pero la prueba es **predicado** en lugar de la igualdad con un objeto; por ejemplo:

```
(MEMBER-IF 'ATOM '((A B) C D)) -> (C D)
(MEMBER-IF 'LISTP '(A B C)) -> NIL
```


Para utilizar las listas de asociación (como tablas), se utiliza la función ASSOC.

(ASSOC clave lista)

La función ASSOC asume que la estructura de **lista** tiene una forma particular:

((clave$_1$ expresión$_1$)
(clave$_2$ expresión$_2$)
(clave$_3$ expresión$_3$)
.
.
.
(clave$_N$ expresión$_N$))

en donde **clave$_I$** es un átomo y **expresión$_I$** cualquier expresión LISP.

La forma de actuar de ASSOC es buscar en **lista** un elemento, una sublista **(clave$_I$ expresión$_I$)** cuyo CAR sea igual a **clave** y retornar ese elemento.

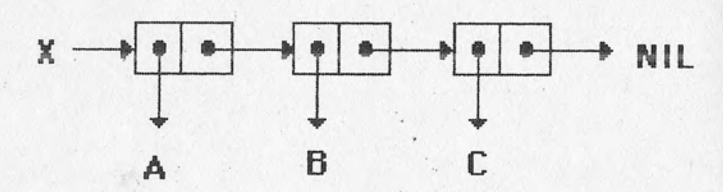
```
(SETQ ORDINALES '((1 PRIMERO)
                  (2 SEGUNDO)
                  (3 TERCERO)) ->
-> ((1 PRIMERO) (2 SEGUNDO) (3 TERCERO))
(ASSOC 2 ORDINALES) -> (2 SEGUNDO)
(ASSOC 'PRIMERO '((PRIMERO A) (SEGUNDO B))) ->
-> (PRIMERO A)
```

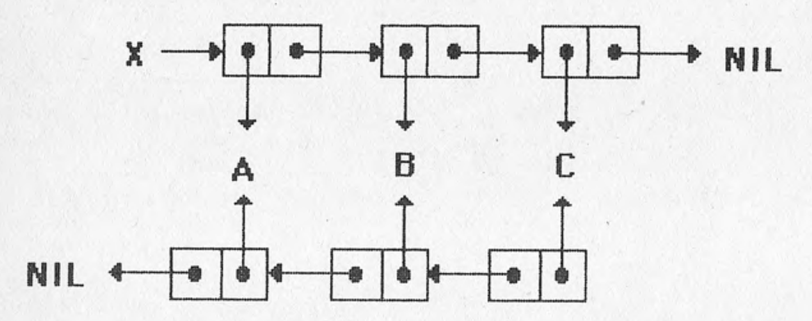
Hay otras dos funciones que no encajan exactamente entre las anteriores, pero que son bastante útiles. Se trata de REVERSE y NREVERSE, que evalúan a una lista con el orden invertido. REVERSE crea una copia y NREVERSE modifica el original. La sintaxis de ambas es:

(REVERSE lista)
(NREVERSE lista)

su uso se aclarará con un ejemplo:

```
(SETQ X '(A B C)) -> (A B C)
(REVERSE X) -> (C B A)
```





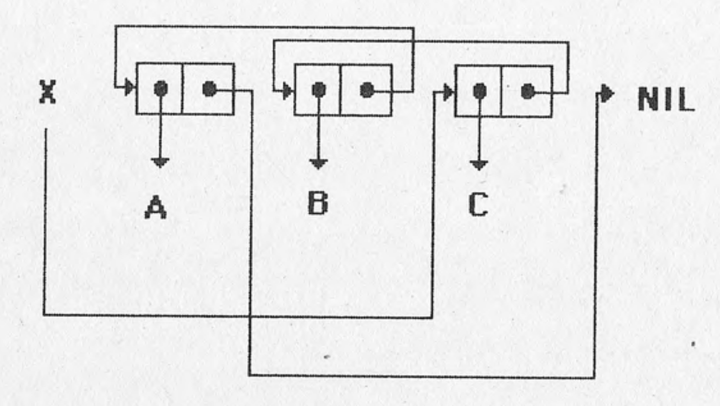


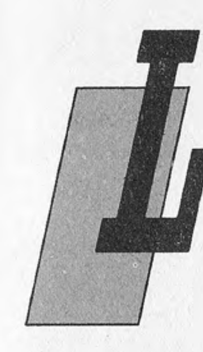
Figura 8.—Comparación de REVERSE y NREVERSE.


```
X -> (A B C)
(NREVERSE X) -> (C B A)
X -> (C B A)
```

En la figura 8 se observan las estructuras celulares correspondientes.

CAPITULO VII

CONDICIONALES



os condicionales son una de las principales estructuras de control de todo lenguaje de programación. Dirigen el flujo de información por el programa mediante la toma de decisiones. El manejo de condicionales en LISP es bastante semejante al de otros lenguajes. En este capítulo se expondrán las principales estructuras condicionales, con abundantes ejemplos.

WHEN

Es la forma más sencilla de tomar decisiones. Como el resto de los condicionales se trata de una forma especial, por lo que tiene sus propias reglas de evaluación. su sintaxis es la siguiente:

(WHEN test forma$_1$ forma$_2$ ... forma$_N$)

Donde:

- test: es la primera forma evaluada. Si retorna NIL, ninguna de las demás es considerada y WHEN resulta NIL. En otro caso, comienza la evaluación del resto de las formas.
- forma$_1$ forma$_2$ ... forma$_N$: Cuando **test** retorna no-NIL son evaluadas secuencialmente; WHEN retorna **forma$_N$**, es decir, la última forma.

Generalmente WHEN se usa para producir efectos laterales de forma condicional (como la transferencia del control a otros

procedimientos) y no suele interesar el resultado de su evaluación. Cuando es este último el principal objetivo es más elegante el empleo de AND, que se tratará posteriormente en este mismo capítulo.

A continuación pueden verse algunos ejemplos sencillos de WHEN:

```
(WHEN (NUMBERP 8) (SETQ A (* 2 3)) (SETQ B (+ 10 20))) ->
30
A -> 6
B -> 30

(WHEN (LISTP 'FOO) (SETQ X (* 4 5)) -> NIL
X -> ¡¡¡ERROR!!! Variable no ligada.
```

el test evaluado NIL y por tanto no evalúan el resto de las formas.

UNLESS

Su estructura es idéntica a la de WHEN, pero funciona de manera contraria. La sintaxis de esta forma especial es la siguiente:

(UNLESS test forma$_1$ forma$_2$ ... forma$_N$)

Donde:

- test: es la primera forma evaluada. Si retorna no-NIL finaliza la evaluación de UNLESS y éste devuelve NIL. Si test retorna NIL comienza la evaluación del resto de las formas.
- forma$_1$ forma$_2$ ... forma$_N$: Se evalúan secuencialmente en caso de que **test** sea NIL. UNLESS retorna el valor de **forma$_N$**.

Así, pues, vemos que ambas formas especiales tienen la estructura:

Condición → acción

en el caso de WHEN la acción se lleva a cabo si la condición es cierta, y en el caso de UNLESS, si la condición es falsa.

Puede aplicarse a UNLESS el mismo comentario que se hacía sobre WHEN en lo referente a la elegancia de su empleo.

Algún ejemplo:

```
X -> ¡¡¡ERROR!!! Variable no ligada.
(UNLESS (LISTP '(FOO BAR)) (SETQ X (* 3 5))) -> NIL
X -> ¡¡¡ERROR!!! Variable no ligada.
(SETQ FOO 7) -> 7
(UNLESS (NUMBERP FOO) (SETQ X (* 3 5)) -> 15
X -> 15
```

IF

Se trata de otra forma especial de tipo condicional, de estructura algo más compleja. Su sintaxis es la siguiente:

(IF test entonces en-otro-caso)

Donde:

- test: es la primera forma evaluada. Si retorna no-NIL se evalúa **entonces**, si retorna NIL se pasa a **en-otro-caso.**
- entonces: primera acción del IF. Si **test** retorna no-NIL es evaluada y el IF retorna el resultado de esta evaluación.
- en-otro-caso: segunda acción del IF. Si **test** da NIL es evaluada y el IF devuelve este resultado. Puede prescindirse de esta forma; en ese caso, si **test** resulta NIL, IF retorna NIL, pero en este supuesto es más conveniente usar WHEN.

Algún ejemplo de IF

```
(IF (SYMBOLP 1) (* 3 3) (+ 2 2)) -> 4
(DEFUN FOO (X)
     (IF (LISTP X) 'LISTA 'ATOMO)) -> FOO
(FOO 4) -> ATOMO
(FOO '(HOLA PEPE)) ->LISTA
(SETQ BAR '(X Y Z)) -> (X Y Z)
(FOO BAR) -> LISTA
```

```
(DEFUN VALOR-ABSOLUTO (X)
      (IF (< X 0) (- X) X)) -> VALOR-ABSOLUTO
(VALOR-ABSOLUTO 3) -> 3
(VALOR-ABSOLUTO -2) -> 2
```

COND

Es la estructura condicional más completa de todas las que vamos a tratar aquí. Su sintaxis es la siguiente:

(COND (test$_1$ consec$_{1\text{-}1}$ consec$_{1\text{-}2}$ ... consec$_{1\text{-}m}$)
(test$_2$ consec$_{2\text{-}1}$ ... consec$_{2\text{-}H}$)
.
.
(test$_N$ consec$_{N\text{-}1}$ ... consec$_{N\text{-}L}$))

Cada una de las listas de la forma **(test$_J$ consec$_{J\text{-}1}$ ... consec$_{J\text{-}K}$)** se denomina **cláusula.** COND evalúa secuencialmente las cláusulas. Para ello examina el **test** de cada una. Si resulta NIL la cláusula es ignorada y se pasa a la siguiente. Cuando un **test** es no-NIL lleva a cabo la evaluación de esa cláusula con la evaluación secuencial de todas las formas **consecuentes.** COND retorna el valor de la última forma evaluada de la cláusula en cuestión sin evaluar las cláusulas restantes.

Así, pues, un COND está compuesto de una serie de parejas **condición-consecuentes** y evalúa los consecuentes correspondientes a la primera condición que sea no-NIL. En el caso de que ninguna de ellas sea no-NIL, COND retorna el valor NIL. Si la cláusula con test no-NIL no tiene ningún consecuente, COND ofrece el valor del test.

Es muy recomendable que la última cláusula de un COND tenga como test T, de manera que sus consecuentes se evalúen siempre que todos los demás tests sean falsos. Suele utilizarse, al menos, la cláusula **(T NIL).**

Unos ejemplos aclararán estas ideas:

```
(DEFUN COMPARA (X Y)
    (COND ((= X Y) 'LOS-NUMEROS-SON-IGUALES)
          ((< X Y) 'EL-PRIMERO-ES-MENOR)
          ((> X Y) 'EL-PRIMERO-ES-MAYOR)  )) -> COMPARA
```



```
(COMPARA 2 3) -> EL-PRIMERO-ES-MENOR
(COMPARA 10 20) -> EL-PRIMERO-ES-MAYOR
(COMPARA 5 5) -> LOS-NUMEROS-SON-IGUALES
```

Las tres primitivas que se han utilizado en el cuerpo de este DEFUN sólo admiten números como argumentos. Por tanto, si se llama a la función COMPARA con argumentos no numéricos, se tendrá un error.

Dado que si las dos primeras condiciones son falsas la tercera ha de ser forzosamente cierta, podría haberse sustituido esa cláusula por (T 'EL-PRIMERO-ES-MAYOR). En otro ejemplo puede quedar más clara la utilidad del empleo de "T" como condición de la última cláusula.

```
(DEFUN CAPITAL-DE (X)
    (COND ((EQUAL X 'FRANCIA) 'PARIS)
          ((EQUAL X 'ESPAÑA) 'MADRID)
          ((EQUAL X 'PERU) 'LIMA)
          ((EQUAL X 'PORTUGAL) 'LISBOA)
          (T 'NO-LO-SE)                 )) -> CAPITAL-DE
(CAPITAL-DE 'PERU) -> LIMA
(CAPITAL-DE 'ESPAÑA) -> MADRID
(CAPITAL-DE 'ARGENTINA) -> NO-LO-SE
```

Siempre que se suministre como argumento un país que no esté entre las condiciones del COND este programa contestará NO-LO-SE.

Funciones lógicas

Son semejantes a los condicionales, pero en ellas lo más interesante es el efecto principal. Corresponden a las conectivas copulativa y disyuntiva de la lógica tradicional.

AND es una macro que resulta NIL si alguno de sus argumentos es falso y -NIL si todos son ciertos. Recuérdese que en LISP la forma de expresar la falsedad es NIL, y cualquier otra expresión,

particularmente T, representa verdad. La sintaxis de AND es la siguiente:

(AND $expr_1$ $expr_2$ ... $expr_N$)

AND evalúa secuencialmente todas las expresiones argumento. Cuando encuentra alguna que es NIL retorna NIL y termina su evaluación, sin evaluar las restantes. Si no encuentra ninguna NIL retorna el valor de **$expr_N$**. Un ejemplo aclarará esto.

```
(AND (LISTP '(A B C)) (LISTP '(FOO BAR)) -> (FOO BAR)

(AND (LISTP '(A B C)) (LISTP 'A)) -> NIL

(SETQ X '(A B C)) -> (A B C)

(AND (LISTP X) (LISTP 9)) -> NIL

Z -> ¡¡¡ERROR!!! Variable no ligada.

Y -> ¡¡¡ERROR!!! Variable no ligada.

(AND (SETQ Z 8) (LISTP 9) (SETQ Y 7)) -> NIL

Z -> 8

Y -> ¡¡¡ERROR!!! Variable no ligada. Las expresiones

posteriores a (LISTP 9) -> NIL no se evalúan.
```

OR da como resultado NIL si todos sus argumentos son NIL; Si hay alguno no-NIL retorna no-NIL. Su sintaxis es semejante a la de AND:

(OR $expr_1$ $expr_2$ ... $expr_N$)

OR evalúa secuencialmente sus argumentos. Cuando encuentra alguno no-NIL retorna este valor y finaliza su evaluación ignorando las restantes expresiones. Si todos son NIL OR resulta NIL. Por ejemplo:

```
(OR (NUMBERP 6) (NUMBERP 7) (NUMBERP 8)) -> T

X -> ¡¡¡ERROR!!! Variable no ligada.

Y -> ¡¡¡ERROR!!! Variable no ligada.

(OR (NUMBERP 'E) (SETQ X 'FF) (SETQ Y 'GG)) -> FF
```



```
X -> FF

Y -> ¡¡¡ERROR!!! Variable no ligada.
```

La última expresión no se ha evaluado.

```
(NOT (LISTP '(A B C))) -> NIL

(NOT (NUMBERP '(A))) -> T
```

NUMBERP es un predicado que ofrece T cuando su argumento es un número y NIL en otro caso.

Un importante complemento de AND y OR es NOT. NOT devuelve T cuando su argumento es NIL, y NIL en cualquier otro caso. Su sintaxis es muy sencilla:

(NOT arg)

CAPITULO VIII

OTRAS ESTRUCTURAS DE CONTROL: SECUENCIAMIENTO E ITERACION

Una de las estructuras de control más rudimentarias es la iteración, que consiste en la ejecución repetida de una serie de procedimientos hasta que se satisface una determinada condición que interrumpe el proceso.

En LISP el método más antiguo de iteración es el empleo de las construcciones **prog.** Con ellas se puede escribir un programa con una filosofía similar a los escritos en FORTRAN o BASIC. En un principio eran la única forma de crear de un modo explícito un bucle de programa; en la actualidad, con la macro DO, que se tratará posteriormente, han quedado superados.

Hay varias construcciones de este tipo: la macro PROG y las formas especiales PROGN, PROG1 y PROG2. Se tratarán todas ellas (con fines didácticos), aunque debe tenerse siempre presente que es más aconsejable utilizar la macro DO.

PROG

Su sintaxis es la siguiente:

(PROG (var_1 var_2 (var_3 forma-$inic_3$) ... var_K)
($forma_1$ $forma_2$ $marca_1$ $forma_3$ ... $marca_M$ $forma_N$))

En donde:

- var_1 ... var_K: son variables que quedan ligadas localmente dentro del PROG. Pueden aparecer solas o dentro de una

lista de la forma (**var$_I$ forma-inic$_I$**), en donde **forma-inic$_I$** representa una forma que, evaluada, da el valor inicial de **var$_I$**. En primer lugar se evalúan secuencialmente todas las **forma-inic$_I$**, a continuación se ligan las variables a los valores de sus respectivas formas. En caso de que una variable carezca de **forma-inic** se liga a NIL. Al terminar de evaluar el PROG desaparecen las ligaduras creadas por él y todas las variables recuperan el valor que tenían. En caso de que no haya ninguna variable debe aparecer la lista vacía.

- **forma$_1$ ... forma$_N$**: Conjunto de formas que constituyen el cuerpo del PROG. Son evaluadas secuencialmente una tras otra. Tras serlo la última termina la evaluación del PROG, que retorna como valor NIL.
- **marca$_1$ ... marca$_M$**: Son símbolos o números enteros que quedan sin evaluar e identifican una posición dentro del grupo de formas que constituyen el cuerpo del PROG.

Entre las formas del PROG es interesante destacar dos que son específicas de esta construcción:

- **GO marca$_I$**: Cuando se alcanza esta expresión el control de la evaluación es transferido a la forma inmediatamente posterior a **marca$_I$**. Equivale a las sentencias GOTO de otros lenguajes de programación. En general no conviene utilizar esta expresión debido a cuestiones de estilo, puesto que dificulta la lectura y entendimiento de los programas.
- **RETURN forma**: Produce el final de la evaluación de PROG. El valor retornado por éste es el resultante de la evaluación de la forma que acompaña a RETURN.

A continuación se muestra un ejemplo práctico de la sintaxis del PROG mediante la función FACTORIAL, que calcula el factorial del número suministrado como argumento.

```
(DEFUN FACTORIAL (N)
    (PROG (NUMERO RESULTADO)
        (SETQ NUMERO N)
        (SETQ RESULTADO 1)
        MARCA
        (WHEN (ZEROP NUMERO) (RETURN RESULTADO))
        (SETQ RESULTADO (* RESULTADO NUMERO))
```



```
        (SETQ NUMERO (- NUMERO 1))
        (GO MARCA) ) ) -> FACTORIAL
```

Este PROG utiliza como variables locales NUMERO y RESULTADO. Inicialmente se ligan a NIL y a continuación se les asignan, respectivamente, los valores 1 y N (el número del que queramos obtener factorial). También podían haberse dado los valores iniciales dentro de la lista de variables que aparece en la cabecera del PROG. A continuación se encuentra un bucle de iteración que calcula en cada pasada uno de los productos del factorial y decrementa NUMERO en una unidad. Al comenzar el bucle se comprueba si NUMERO es igual a cero, en cuyo caso se termina el PROG retornando RESULTADO como valor.

Como se puede observar, salvando las distancias impuestas por la sintaxis de cada lenguaje, la estructura del PROG es similar a la de un programa en BASIC, por ejemplo.

Uno de los motivos que desaconsejan el uso del PROG es que la asignación de valores iniciales a las variables su actualización, y las comprobaciones de fin de bucle, pueden estar dispersas a lo largo del cuerpo. Por otra parte puede haber múltiples señales y formas GO. Todo esto dificulta el seguimiento de la evaluación del procedimiento y hace preferible utilizar los DO, en donde cada una de las partes antes citadas tiene una posición definida dentro de la estructura. Puede demostrarse que todo programa que utilice PROG puede reescribirse utilizando DO.

LOOP

Es la estructura más simple de iteración. Se limita a ejecutar repetidamente su cuerpo sin tener control sobre variables. Su sintaxis es

(LOOP forma$_1$ forma$_2$ ... forma$_N$)

Sus formas deben ser listas; en futuras extensiones de Common LISP está prevista la posibilidad de incluir átomos.

LOOP evalúa secuencialmente cada forma y, tras terminar con la última, vuelve a comenzar con la primera. RETURN permite terminar este proceso retornando el valor especificado en su argumento.

PROGN, PROG1, PROG2

Estas construcciones no tienen demasiada relación con el PROG tratado antes. La forma de trabajo de las tres es bastante parecida: se limitan a evaluar, una tras otra, todas las formas contenidas en su cuerpo, diferenciándose en el valor que retornan. PROGN retorna el valor de la última forma evaluada, PROG1 el de la primera y PROG2 el de la segunda.

Sus sintaxis son:

(PROGN forma$_1$ forma$_2$ ... forma$_N$)
(PROG1 forma$_1$ forma$_2$ ... forma$_N$)
(PROG2 forma$_1$ forma$_2$ ... forma$_N$)

No son demasiado útiles, puesto que hay otros métodos de evaluar secuencialmente una serie de formas, como DEFUN, LET o DO. Todos estos utilizan un PROGN de forma implícita, puesto que evalúan todas sus formas y retornan el valor de la última.

LET

Es una forma especial que tiene el mismo comportamiento que el PROGN, con la facilidad adicional de permitir la utilización de variables locales.

Su sintaxis es la siguiente:

(LET ((var$_1$ valor$_1$)
(var$_2$ valor$_2$)
...
(var$_N$ valor$_N$))
cuerpo-de-LET)

En primer lugar evalúa las expresiones **valor$_1$, valor$_2$,** hasta **valor$_N$,** de forma secuencial y va almacenando los resultados para después ligar simultáneamente todas las variables a sus respectivos valores (se dice que las variables si ligan *en paralelo).* La ligadura se mantiene durante la ejecución de las formas contenidas en el cuerpo, desapareciendo cuando ésta termina.

La lista **(var$_I$ valor$_I$)** puede ser reemplazada simplemente por **var$_I$**, en cuyo caso el valor inicial de esta variable es NIL; sin embargo, es aconsejable, por razones de estilo, que siempre aparezca la lista de la variable con su valor.

El **cuerpo-de-LET** está compuesto por una serie de formas que son ejecutadas secuencialmente. Como valor devuelto por la evaluación de LET se retorna el resultado de la última forma del cuerpo; se puede decir que el cuerpo de LET es un PROGN implícito.

LET es particularmente útil para almacenar el resultado de una operación que debe repetirse varias veces a lo largo del proceso; por ejemplo, la función ARITMETICA realiza la suma, resta, multiplicación y división de sus dos argumentos:

```
(DEFUN ARITMETICA (X Y)
    (LET ((LISTA (LIST X Y)))
        (SETF SUM (APPLY #'+ LISTA))
        (SETF DIF (APPLY #'- LISTA))
        (SETF PROD (APPLY #'* LISTA))
        (SETF COC (APPLY #'/ LISTA)))) -> ARITMETICA
(ARITMETICA 8 4) -> 2
SUM -> 12
DIF -> 4
PROD -> 32
COC -> 2
(BOUNDP LISTA) -> NIL
```

En este ejemplo sólo se evalúa una vez (LIST X Y), almacenándose su resultado en LISTA; de no haber hecho esto, deberíamos haber repetido dicha operación con cada una de las formas SETF, lo que redundaría en una mayor ineficencia del procedimiento. Otra posibilidad sería almacenar el resultado de (LIST X Y) en LISTA mediante un SETF; sin embargo, sigue siendo más conveniente usar LET, puesto que la alternativa deja globalmente ligada la variable LISTA, ocupando una posición de memoria de forma permanente. Esto, en principio, no tiene ninguna utilidad y resulta menos elegante.

*LET**

Esta forma especial se diferencia de LET en que las ligaduras de las variables con sus valores no se hace en paralelo, sino de forma secuencial. En primer lugar se evalúa **forma$_1$** y **var$_1$** se liga a su resultado. Con las restantes variables se va procediendo del mismo modo, una tras otra. Esta forma de actuar presenta la ventaja de que pueden utilizarse los valores de las variables previa-

mente ligadas en las formas que dan valor a las variables siguientes. La sintaxis de LET* es idéntica a la de LET.

*DO y DO**

Estas macros son las construcciones de uso general para iteración. Como ya se comentó anteriormente, DO permite realizar los mismos procedimientos que los PROG, con la ventaja de que tienen zonas definidas para actualizar las variables cada vez que se realiza un ciclo de iteración y para realizar la comprobación de fin de bucle. Su estructura está más ordenada.

La sintaxis de DO es la siguiente (la de DO* es exactamente igual, cambiando DO por DO* en la cabecera):

(DO ((var_1 valor-$inic_1$ $actual_1$)
(var_2 valor-$inic_2$ $actual_2$)
...
(var_N valor-$inic_N$ $actual_N$))
(test-de-fin expresiones-fin)
cuerpo)

Puede decirse que hay tres partes en esta construcción:

— La primera contiene la lista de parámetros junto con sus valores iniciales y su actualización.
— La segunda determina cuándo finaliza el proceso de iteración y qué se hace al terminar.
— La tercera está ocupada por formas que se evalúan secuencialmente.

En primer lugar se evalúan todas las formas **valor-$inic_I$** y se asignan en paralelo sus resultados a las variables locales **var_I**. En caso de omisión de algún **valor-$inic_I$** (lo que implica también la omisión de **$actual_I$**) la variable correspondiente recibe NIL como valor inicial. Al igual que se dijo en el caso de LET conviene dejar explícito el valor inicial de todas las variables.

Una vez ligadas todas las variables locales se realiza la comprobación especificada en **test-de-fin;** en el caso de que esta forma evalúe a NIL se continúa con las expresiones que la siguen, evaluándolas en orden como un PROGN implícito. El valor de la última es retornado como resultado del DO. Si **test-de-fin** no es NIL entonces se comienza a evaluar el cuerpo del DO, el cual admite las mismas expresiones que el del PROG. Los valores de las formas no son retornados, por lo que sólo se dejan sentir sus efectos laterales. Si se encuentra una expresión que comienza por RETURN se termina inmediatamente el DO, cuyo valor retornado será el del argumento de dicha expresión.

Al terminar de evaluarse el cuerpo finaliza un ciclo de iteración. En ese instante se actualizan las variables con la asignación, también en paralelo, de los valores obtenidos en la evaluación de las formas **$actual_I$**. A continuación se vuelve a comprobar, con **test-de-fin,** si debe finalizar el proceso iterativo. En función del resultado que se obtenga se sigue uno de los dos caminos alternativos anteriormente descritos.

Si se tratara de un DO* las variables serían ligadas a sus valores iniciales y actualizadas de forma secuencial.

La función FACTORIAL puede ser redefinida utilizando un DO:

```
(DEFUN FACTORIAL (N)
    (DO ((RESULTADO 1 (* RESULTADO NUMERO))
        (NUMERO N (- NUMERO 1)))
        ((ZEROP NUMERO) RESULTADO)))
```

En este caso no existe cuerpo y el valor del factorial se calcula a través de la actualización de la variable RESULTADO. Obsérvese cómo, mediante esta construcción, la función FACTORIAL resulta mucho más compacta.

El que las variables se liguen y actualicen en paralelo o en serie puede influir grandemente en el valor que se les asigna y ser fuente de errores. En el ejemplo, tal como está escrito, no hay problemas para utilizar DO* en lugar de DO. Pero si se invierte el orden de asignación dejándolo como figura a continuación, sólo puede utilizarse correctamente con DO:

```
(DEFUN FACTORIAL (N)
    (DO ((NUMERO N (- NUMERO 1))
    (RESULTADO 1 (* RESULTADO NUMERO)))
    ((ZEROP NUMERO) RESULTADO)))
```

En efecto, si se utiliza DO* la variable NUMERO es actualizada antes de obtenerse su producto con RESULTADO.

CAPITULO IX

OPERADORES APLICATIVOS

El uso de operadores aplicativos es un tipo de iteración característico de LISP. Permiten aplicar sucesivamente una función a cada elemento de una *secuencia* (concepto que, a efectos de este libro, se considera equivalente al de lista). El resultado es una secuencia que contiene los valores retornados por las sucesivas aplicaciones de la función.

Sólo presentaremos los operadores aplicativos sobre listas, ilustrándolos con algunos ejemplos sencillos.

Las construcciones que se ofrecen a continuación tienen diferentes modos de seleccionar los elementos de la lista y de tratar los resultados obtenidos de la aplicación de la función. Sus sintaxis son semejantes: consisten en el nombre del operador utilizado seguido por el de la función que va a aplicarse y una o varias listas que contienen los argumentos de la función; así:

(MAPCAR función $\textbf{lista}_1$ $\textbf{lista}_2$ **...** $\textbf{lista}_N$**)**
(MAPLIST función $\textbf{lista}_1$ $\textbf{lista}_2$ **...** $\textbf{lista}_N$**)**
(MAPC función $\textbf{lista}_1$ $\textbf{lista}_2$ **...** $\textbf{lista}_N$**)**
(MAPL función $\textbf{lista}_1$ $\textbf{lista}_2$ **...** $\textbf{lista}_N$**)**
(MAPCAN función $\textbf{lista}_1$ $\textbf{lista}_2$ **...** $\textbf{lista}_N$**)**
(MAPCON función $\textbf{lista}_1$ $\textbf{lista}_2$ **...** $\textbf{lista}_N$**)**

En la posición de **función** no puede aparecer el nombre de una macro o de una forma especial. La función puede ser una de las definidas por el usuario con nombre, una expresión LAMBDA o una primitiva del sistema.

El número de listas que sigue a la función debe coincidir con el número de argumentos que requiera la misma. Esta tomará el primero de ellos de los elementos de **lista$_1$**, el segundo de los de **lista$_2$** y así sucesivamente hasta completar el número total de argumentos. Debe procurarse que la longitud de todas las listas sea igual; en caso contrario la iteración termina cuando se llegue al final de la lista más corta, ignorándose los elementos sobrantes de las otras.

- MAPCAR: opera con cada uno de los elementos de la lista de forma sucesiva. La primera vez con el **CAR**, la segunda con el **CADR**, la tercera con el **CADDR**, etc.

Retorna los resultados de todas las aplicaciones de la función combinados del mismo modo que si se les hubiera aplicado la función LIST.

```
(MAPCAR #'NULL '(3 () 0 NIL FOO) -> (NIL T NIL T NIL)

(MAPCAR #'LIST '(A 3 T)

                '(B 4 NIL)) -> ((A B) (3 4) (T NIL))
```

Nótese el símbolo #' situado delante del nombre de la función y el apóstrofe delante de las listas.

- MAPLIST: aplica la función a la lista completa y a sus sucesivos **CDR.** Los resultados los retorna de la misma manera que MAPCAR.

```
(MAPLIST 'CONS '(A B C)

               '(FOO BAR BAZ)) -> (((A B C) FOO BAR BAZ)

                                   ((B C) BAR BAZ)

                                   ((C) BAZ))
```

- MAPC y MAPL: se comportan, respectivamente, como **MAPCAR** y **MAPLIST** pero, a diferencia de éstos, retornan tras su evaluación la primera lista de argumentos. Su utilidad principal reside, por lo tanto, en los efectos laterales de la función aplicada.
- MAPCAN y MAPCON: también coinciden con **MAPCAR** y **MAPLIST,** pero en este caso combinan los resultados de la función usando **NCONC** en lugar de **LIST.** Los resultados cuyo valor sea NIL son eliminados de la lista; ésta puede tener, por tanto, una cantidad variable de elementos.


```
(MAPCAN #'NULL '(3 () 0 NIL FOO) -> (T T)

(MAPCAN #'LIST '(A 3 T)

               '(B 4 NIL)) -> (A B 3 4 T NIL)

(MAPCON 'CONS '(A B C)

              '(FOO BAR BAZ)) -> ((A B C) FOO BAR BAZ (B

                                  C) BAR BAZ (C) BAZ)
```

Al emplear los operadores aplicativos suele ser útil recurrir a las expresiones LAMBDA. Obsérvese el siguiente ejemplo:

```
(DEFUN CUENTA-NUM (LISTA)

  (APPLY #'+

      (MAPCAN #'(LAMBDA (X)

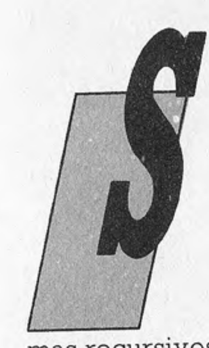
        (IF (NUMBERP X) 1 NIL)) LISTA))) -> CUENTA-NUM

(CUENTA-NUM '(A 2 R 5 10 FOO)) -> 3
```

La función CUENTA-NUM retorna la cantidad de números contenida en una lista. La expresión LAMBDA evalúa a 1 si su argumento es un número y a NIL si no lo es. Este último valor es eliminado al construirse la lista a la que evalúa MAPCAN. Por último, la función "+" suma los elementos de esa lista.

CAPITULO X

RECURSION



e denomina recursión (o recursividad) al hecho de que una función o procedimiento se llame a sí mismo en el curso de su evaluación. Esta posibilidad aporta, en muchos casos, una alternativa a las estructuras de control tradicionales. La utilización de la recursión suele dar como resultado unos procedimientos más compactos y simples. En este capítulo daremos unas breves indicaciones sobre la estructura de los programas recursivos, para después ilustrar extensamente este concepto apoyándonos en varios ejemplos.

LISP está construido básicamente sobre la idea de recursión. La lista, que es la principal estructura de datos, tiene una definición recursiva, como ya se vio en el capítulo 2. Por este mismo motivo, el mismo bucle READ-EVAL-PRINT está fundamentado también en este concepto. Al estar LISP así concebido se hace muy sencilla la definición y el empleo de funciones recursivas. Por otra parte, la posibilidad de emplear *ligaduras múltiples* sobre una misma variable, que se enmascaran automáticamente cada vez que se evalúa una función, evita al programador gran parte del control de la transferencia de información entre los procedimientos.

Estructura general de la recursión

Un problema susceptible de ser tratado mediante recursión ha de reunir tres características:

- Su resolución ha de poder ser descompuesta en la resolución de una etapa y la del resto del problema;
- la resolución del resto ha de ser análoga a la del problema completo;
- debe existir un estado límite del problema cuya resolución sea inmediata y que señala el fin del proceso recursivo.

Un ejemplo típico de este tipo de problemas es el cálculo del factorial de un número. En efecto:

— Su resolución se puede descomponer en el cálculo del factorial de número menos uno y su multiplicación por el propio número;
— la resolución del factorial de un número menos uno es análoga a la del factorial del número;
— existe un estado límite (factorial de uno) que es uno.

Estas características llevan inmediatamente a la definición de un algoritmo recursivo:

1. Si se está en el caso límite → solución inmediata.
2. En otro caso, separar una etapa y resolver el resto.

Aplicado al factorial, y expresado en LISP, esto resultaría:

```
(DEFUN FACT (N)
    (COND ((EQUAL N 1) 1)
          (T (* N (FACT (- N 1))))))
```

Las dos cláusulas del COND reflejan la estructura antes comentada. La primera comprueba la condición límite, dando la solución inmediata si aquélla se cumple. La segunda cláusula, es la propiamente recursiva: siempre que no se verifique la condición límite, descompone el problema y aplica el mismo procedimiento a la resolución del resto. Es decir, calcula el factorial de N-1 y lo multiplica por N. Compárese este algoritmo con el procedimiento iterativo tratado en el capítulo 8.

En la figura 1 se muestra una ejecución de (FACT 5) con la ayuda de la macro TRACE. Esta es una ayuda a la depuración de programas LISP que muestra las llamadas a las funciones con sus argumentos y las salidas con los resultados de la evaluación. Se tratará más ampliamente en un capítulo posterior.

Otro ejemplo sencillo de problema de estructura recursiva es el paso de un número entero en sistema decimal al sistema bina-


```
* (DEFUN FACT (N)
        (COND ((EQUAL N 1) 1)
              (T (* N (FACT (- N 1))))))
FACT
* (TRACE FACT)
T
* (FACT 5)
Entering: FACT, Argument list: (5)
 Entering: FACT, Argument list: (4)
  Entering: FACT, Argument list: (3)
   Entering: FACT, Argument list: (2)
    Entering: FACT, Argument list: (1)
    Exiting: FACT, Value: 1
   Exiting: FACT, Value: 2
  Exiting: FACT, Value: 6
 Exiting: FACT, Value: 24
Exiting: FACT, Value: 120

 120
 *
```

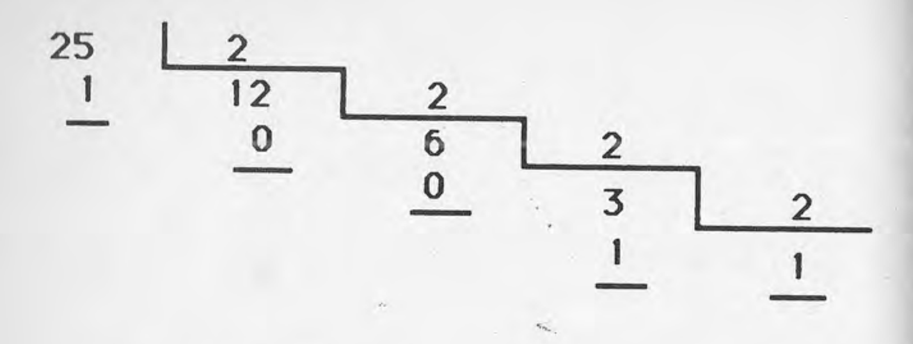
Figura 1.—Ilustración del funcionamiento recursivo de FACT.

rio. La operación "normal" para hacer esto consiste en ir dividiendo por 2 sucesivamente el número y los cocientes obtenidos. Los restos calculados y el último cociente, en orden inverso, expresan el número en base 2. En la figura 2 se muestra el algoritmo "manual".

El problema se puede descomponer en obtener el resto de dividir el número entre 2 y pasar a base 2 el cociente obtenido. La condición límite es que el número a convertir sea menor que 2; en ese caso la solución es trivial y es el mismo número (0 ó 1).

La función LISP correspondiente es:

```
(DEFUN BINARIO (N)
    (COND ((< N 2) N)
          (T (+ (* (BINARIO (TRUNCATE N 2)) 10) (MOD N 2)))))
```



$$25_{10} = 11001_2$$

Figura 2.—Algoritmo para paso de sistema de numeración decimal a sistema binario.

La primera cláusula del COND comprueba la condición límite. Si el número es menor que 2 su expresión binaria es él mismo. La segunda descompone el problema y vuelve a llamar a la función BINARIO. El cuerpo de esta segunda cláusula incluye varias funciones que no se han explicado: TRUNCATE obtiene el cociente entero de N y 2; MOD calcula el resto de la división.

Con esto el proceso se desarrolla de la siguiente forma: (BINARIO (TRUNCATE N 2)) devuelve la expresión binaria del cociente. Esta expresión se multiplica por 10, pues la lógica numérica del ordenador es decimal, y se le suma el resto. En la figura 3, la macro TRACE ilustra el funcionamiento de (BINARIO 25).

Un ejemplo de recursión con listas

Vamos a plantear a continuación una definición de la intersección de listas, basada en la recursión. Se trata de una función que toma como argumentos dos listas y retorna una tercera con todos los elementos comunes de las primeras.

El procedimiento que sigue consiste en estudiar si el primer elemento de la primera lista es miembro de la segunda. En caso afirmativo se incorpora a la lista que retornará como resultado. Posteriormente se halla la intersección del resto de la primera lista con la segunda. La condición límite es que la primera lista sea


```
* (DEFUN BINARIO (N)
        (COND ((< N 2) N)
              (T (+ (* (BINARIO (TRUNCATE N 2)) 10) (MOD N 2)))))
BINARIO
* (TRACE BINARIO)
T
* (BINARIO 25)
Entering: BINARIO, Argument list: (25)
 Entering: BINARIO, Argument list: (12)
  Entering: BINARIO, Argument list: (6)
   Entering: BINARIO, Argument list: (3)
    Entering: BINARIO, Argument list: (1)
    Exiting: BINARIO, Value: 1
   Exiting: BINARIO, Value: 11
  Exiting: BINARIO, Value: 110
 Exiting: BINARIO, Value: 1100
Exiting: BINARIO, Value: 11001

11001
*
```

Figura 3.—Paso del sistema decimal al binario mediante una función recursiva.

vacía, en cuyo caso la intersección es también vacía. La expresión en LISP de este algoritmo es la siguiente:

```
(DEFUN INTERSECCION (X Y)
    (COND ((NULL X) NIL)
          ((MEMBER (CAR X) Y)
           (CONS (CAR X) (INTERSECCION (CDR X) Y)))
          (T (INTERSECCION (CDR X) Y))))
```

Como siempre, la primera cláusula del COND comprueba la condición límite. La segunda incluye en el resultado al primer elemento de "X" si pertenece a "Y" y vuelve a llamar a la función INTERSECCION dando como argumento el resto de "X". La tercera cláusula también es una llamada recursiva a INTERSECCION, con el mismo argumento, despreciándose en este caso el primer elemento. La figura 4 ilustra el funcionamiento de (INTERSECCION '(A B C D) '(A C G)).

```
* (DEFUN INTERSECCION (X Y)
       (COND ((NULL X) NIL)
             ((MEMBER (CAR X) Y) (CONS (CAR X) (INTERSECCION (CDR X) Y)))
             (T (INTERSECCION (CDR X) Y))))
INTERSECCION
* (TRACE INTERSECCION)
T
* (INTERSECCION '(A B C D) '(A C G))
Entering: INTERSECCION, Argument list: ((A B C D) (A C G))
 Entering: INTERSECCION, Argument list: ((B C D) (A C G))
  Entering: INTERSECCION, Argument list: ((C D) (A C G))
   Entering: INTERSECCION, Argument list: ((D) (A C G))
    Entering: INTERSECCION, Argument list: (NIL (A C G))
    Exiting: INTERSECCION, Value: NIL
   Exiting: INTERSECCION, Value: NIL
  Exiting: INTERSECCION, Value: (C)
 Exiting: INTERSECCION, Value: (C)
Exiting: INTERSECCION, Value: (A C)

(A C)
*
```



Figura 4.—INTERSECTION: función recursiva que calcula la intersección de dos listas.

Manejo recursivo de listas anidadas

El siguiente procedimiento de ejemplo es una ampliación de MAPCAR que tiene como fin aplicar una función a todos los elementos terminales de un conjunto de listas anidadas. Por ejemplo, si se aplica NUMBERP (predicado que evalúa a T si su argumento es un número y a NIL en otro caso) a:

```
((FOO 3) (A (B C)) BAR  3 (4 5) BAZ)
```

se obtendría como resultado:

```
((NIL T) (NIL (NIL NIL)) NIL T (T T) NIL)
```

La condición de terminación es que el argumento sea un átomo. En ese caso la función se le aplica directamente. Si no es un átomo debe ser una lista y se aplica todo el procedimiento a cada uno de los elementos de la lista que, a su vez, pueden ser átomos o listas. En LISP quedaría:

```
(DEFUN MAP-ARBOL (FUN EXPR)
   (COND ((ATOM EXPR) (FUNCALL FUN EXPR))
     (T (MAPCAR #'(LAMBDA (X) (MAP-ARBOL FUN X)) EXPR))))
```


La primera cláusula comprueba y resuelve el caso límite. La segunda aplica MAP-ARBOL a todos los elementos de EXPR por medio de un MAPCAR. La función FUNCALL realiza una llamada a la función ligada a FUN, empleando como argumento EXPR. Es semejante a la función APPLY ya comentada; se diferencia de ella en que sus argumentos no están contenidos en una lista. En la figura 5 se detalla la aplicación de MAP-ARBOL al ejemplo anterior.

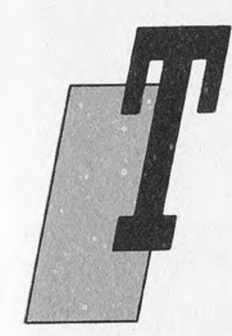
```
*(DEFUN MAP-ARBOL (FUN EXPR)
       (COND ((ATOM EXPR) (FUNCALL FUN EXPR))
             (T (MAPCAR #'(LAMBDA (X) (MAP-ARBOL FUN X)) EXPR))))
MAP-ARBOL
*(TRACE MAP-ARBOL)
T
*(MAP-ARBOL 'NUMBERP '((FOO 3) (A (B C)) BAR 3 (4 5) BAZ))
Entering: MAP-ARBOL, Argument list: (NUMBERP ((FOO 3) (A (B C)) BAR 3 (4 5) BAZ))
 Entering: MAP-ARBOL, Argument list: (NUMBERP (FOO 3))
  Entering: MAP-ARBOL, Argument list: (NUMBERP FOO)
  Exiting: MAP-ARBOL, Value: NIL
  Entering: MAP-ARBOL, Argument list: (NUMBERP 3)
  Exiting: MAP-ARBOL, Value: T
 Exiting: MAP-ARBOL, Value: (NIL T)
 Entering: MAP-ARBOL, Argument list: (NUMBERP (A (B C)))
  Entering: MAP-ARBOL, Argument list: (NUMBERP A)
  Exiting: MAP-ARBOL, Value: NIL
  Entering: MAP-ARBOL, Argument list: (NUMBERP (B C))
   Entering: MAP-ARBOL, Argument list: (NUMBERP B)
   Exiting: MAP-ARBOL, Value: NIL
   Entering: MAP-ARBOL, Argument list: (NUMBERP C)
   Exiting: MAP-ARBOL, Value: NIL
  Exiting: MAP-ARBOL, Value: (NIL NIL)
 Exiting: MAP-ARBOL, Value: (NIL (NIL NIL))
 Entering: MAP-ARBOL, Argument list: (NUMBERP BAR)
 Exiting: MAP-ARBOL, Value: NIL
 Entering: MAP-ARBOL, Argument list: (NUMBERP 3)
 Exiting: MAP-ARBOL, Value: T
 Entering: MAP-ARBOL, Argument list: (NUMBERP (4 5))
  Entering: MAP-ARBOL, Argument list: (NUMBERP 4)
  Exiting: MAP-ARBOL, Value: T
  Entering: MAP-ARBOL, Argument list: (NUMBERP 5)
  Exiting: MAP-ARBOL, Value: T
 Exiting: MAP-ARBOL, Value: (T T)
 Entering: MAP-ARBOL, Argument list: (NUMBERP BAZ)
 Exiting: MAP-ARBOL, Value: NIL
Exiting: MAP-ARBOL, Value: ((NIL T) (NIL (NIL NIL)) NIL T (T T) NIL)

((NIL T) (NIL (NIL NIL)) NIL T (T T) NIL)
*
```

Figura 5.—MAP-ARBOL. Aplicación de una función a un árbol expresado en forma de lista.

CAPITULO XI

LISTAS DE PROPIEDADES Y ESTRUCTURAS



Tanto las listas de propiedades como las estructuras son dos construcciones de LISP que permiten disponer la información en una forma que se asemeja a la de un registro en una base de datos. En ambos casos se asocia a un símbolo un conjunto de componentes que pueden ser cualquier objeto LISP.

Listas de propiedades

Ya se comentó en el capítulo 2 que un símbolo puede tener propiedades. Estas se almacenan en una lista asociada al mismo con un número par de componentes y en la cual, comenzando a contar desde cero, los elementos pares son símbolos que actúan como nombres de las propiedades, y los elementos impares son los valores de éstas. Por ejemplo, el símbolo PERRO puede tener la propiedad PATAS, cuyo valor es 4, también puede tener la propiedad RAZA, con valor CHIHUAHUA. En este caso la lista de propiedades sería algo así:

(PATAS 4 RAZA CHIHUAHUA).

Cada propiedad debe ser única, es decir, no puede haber más que un solo valor y un solo indicador de nombre para cada propiedad. De acuerdo con esto, dado un símbolo y un indicador se obtendrá el valor asociado.

Cuando se crea un símbolo por primera vez su lista de propiedades está vacía. En LISP hay funciones para añadir propieda-

des a un símbolo y para modificar o consultar los valores de las ya existentes.

Funciones de acceso a la lista de propiedades

(GET símbolo indicador)

GET busca en la lista de propiedades de **símbolo** un nombre de alguna de ellas que sea **EQ** a **indicador.** Si lo encuentra, retorna el valor de la misma, en caso contrario retorna NIL. Debe advertirse que no es posible distinguir entre una propiedad inexistente y una cuyo valor es NIL.

(GETF lugar indicador)

Se comporta exactamente igual que **GET** con la diferencia de que **GETF** admite en **lugar** una expresión que retorne un símbolo como valor.

Supóngase que el símbolo PAJARO tiene la siguiente lista de propiedades: (TIPO CANARIO MESES 24 CRIAS NIL), entonces:

```
(GET 'PAJARO 'TIPO)  -> CANARIO

(GET 'PAJARO 'MESES) -> 24

(GET 'PAJARO 'COLOR) -> NIL


(SETF 'AVE (LIST 'PAJARO)) --> (PAJARO)

(GETF (CAR AVE) 'CRIAS)    -> NIL
```

Las funciones **GET** y **GETF** se denominan funciones de acceso, y no de lectura, porque lo que hacen es acceder de forma efectiva a la propiedad señalada por **indicador.** En consecuencia es posible actualizarla, o crearla si no existe, mediante la macro **SETF.** Así, siguiendo con el ejemplo de CANARIO, se obtiene:

```
(SETF (GET 'PAJARO 'COLOR) 'BLANCO) -> BLANCO

(GET 'PAJARO 'COLOR)                -> BLANCO


(SETF (GETF (CAR AVE) 'CRIAS) 4) -> 4

(GET 'PAJARO 'CRIAS)             -> 4
```


Funciones de borrado

Actúan destructivamente con la lista de propiedades, de la cual eliminan tanto el indicador como el valor de la propiedad especificada.

(REMPROP símbolo indicador)

Elimina de la lista de propiedades de **símbolo** aquella cuyo nombre es **EQ** a **indicador.** Si la propiedad denominada **símbolo** existe retorna un valor no-NIL, si no existiera retornaría NIL.

(REMF lugar indicador)

Se diferencia de **REMPROP** sólo en que **lugar** debe ser una forma cuyo valor sea un símbolo.

Siguiendo con PAJARO, su lista de propiedades es ahora:

(TIPO CANARIO MESES 24 CRIAS 4 COLOR BLANCO)

Si se hace

```
(REMPROP 'PAJARO 'COLOR) -> non-NIL

(GET 'PAJARO 'COLOR)     -> NIL
```

La lista quedaría así:

(TIPO CANARIO MESES 24 CRIAS 4)

Utilizando **REMF** con una expresión en **lugar:**

```
(REMF (CAR AVE) 'PATAS) -> NIL        Puesto que no existe

                                      la propiedad PATAS.

(REMF 'PAJARO 'MESES)    -> non-NIL
```

Y la lista es ahora:

(TIPO CANARIO CRIAS 4)

Estructuras

La lista de propiedades permite asociar a un símbolo una serie de características. El conjunto formado por éste y sus propie-

dades puede representar de manera abstracta a una entidad real. El ejemplo con el que se ha trabajado es una muestra clara de esto: se han realizado operaciones con objetos LISP, para los cuales el nombre no tiene más sentido que el de poder distinguir a unos de otros. Sin embargo, de una forma intuitiva se tiende a asociar el símbolo denominado PAJARO con el concepto abstracto que se tiene de un tipo de animales que se identifica por el nombre pájaro.

En Common LISP, la tarea de crear una construcción que permita asociar fácilmente objetos LISP y conceptos abstractos se ve simplificada por el sistema de estructuras.

Una estructura no es más que una agregación de datos que puede identificarse mediante un nombre. Una vez que ha sido declarada como tal mediante la macro **DEFSTRUCT,** la estructura se considera como un nuevo tipo de datos definido por el usuario. Las estructuras así definidas en LISP son semejantes a las de PL/I o a los registros de PASCAL.

Debe tenerse siempre presente la diferencia entre una estructura concreta, que al ser declarada se convierte en un nuevo tipo de datos, y una ocurrencia determinada de esa estructura, que no es más que un objeto LISP cuyo tipo de datos es la estructura anterior. Esto se verá más claro con un ejemplo que aparece posteriormente.

Supóngase que es necesario escribir un programa que trate, por ejemplo, sobre perros. De cada perro es interesante saber su nombre, raza, edad y a quién pertenece. Hay varias formas de unir todos estos elementos:

— Crear un conjunto de listas, una para cada perro, cuyos elementos sean listas de dos átomos, siendo el primero el nombre de la característica que se desea saber, y el segundo, el valor de la misma. Es decir, construir una lista de asociación y emplearla como una tabla. Así:

  ((NOMBRE OSCAR) (RAZA MASTIN) (EDAD 4) (PERTEN-A LOLA))

— Utilizar listas también, pero omitiendo el nombre de la característica e identificando el valor de cada una por la posición que ocupe en la lista; por ejemplo, el primero es el nombre del perro, el segundo su raza y el tercero y el cuarto su edad y el nombre de su dueño, respectivamente. Así, con el caso anterior sería:

  (OSCAR MASTIN 4 LOLA)

— Emplear la lista de propiedades; los nombres de las características serían los indicadores. En el caso del ejemplo:

  (NOMBRE OSCAR RAZA MASTIN EDAD 4 PERTEN-A LOLA)

— Definir la estructura PERRO mediante **DEFSTRUCT** y utilizarla para crear una estructura para cada perro.

Cualquiera de estas formas y otras muchas son válidas, puesto que permiten almacenar información y acceder a ella. Sin embargo, la más potente y sencilla de todas es la que recurre a las estructuras, como se verá a continuación.

La sintaxis de **DEFSTRUCT** es la siguiente:

(DEFSTRUCT (nombre)
(nom-atributo$_1$ valor-omis$_1$)
(nom-atributo$_2$ valor-omis$_2$)
...)

En donde **nombre** y **nom-atributo$_I$** son símbolos que representan el nombre del nuevo tipo de datos y el de su atributo i-ésimo; **valor-omis$_I$** es una forma que se evalúa cada vez que se crea una estructura y cuyo valor sirve como valor inicial del atributo al que acompaña. Puede dejarse sin especificar, en cuyo caso el contenido inicial del atributo queda indeterminado y dependiente de la implantación. El valor que retorna es **nombre.**

En el caso del ejemplo se tendría:

```
(DEFSTRUCT (PERRO)
           (RAZA 'DESCONOCIDA)
           (EDAD 'DESCONOCIDA)
           (PERTEN-A 'NADIE))    -> PERRO
```

Lo que se ha hecho aquí es definir el tipo de datos PERRO, que es una estructura con su nombre y una serie de atributos. Estos, por omisión, tienen unos valores concretos que se especifican en la definición de la estructura.

Los efectos laterales de la evaluación de **DEFSTRUCT** son los siguientes:

- Crea un nuevo tipo de datos denominado **nombre;**
- define una función constructora denominada **MAKE-nom-**

bre, que no admite argumentos y que crea una estructura del tipo **nombre;**

- define una función de acceso a cada atributo denominada **nombre-nom-atributo,** cuyo argumento debe ser un objeto de tipo **nombre.**
- define un predicado denominado **nombre-p,** que retorna T si su argumento es un dato de tipo **nombre** y NIL en caso contrario;
- define una función de copia denominada **COPY-nombre,** cuyo argumento es un objeto de tipo **nombre** y que crea una copia del mismo.

Entre los efectos laterales está la creación de una función que construye objetos del tipo estructura denominada **nombre.** Como ya se dijo antes, es importante diferenciar el tipo estructura de los objetos LISP pertenecientes a ese tipo. Aún hay más, puede haber tantos tipos de datos estructura como se desee. Todos ellos deberán ser declarados y su **nombre** debe ser diferente del de cualquier otro. En la declaración de cada uno de ellos se creará el conjunto de funciones que se pueden aplicar a su tipo de objetos LISP.

La macro **SETF** acepta las funciones de acceso, con lo cual puede ser utilizada para modificar el contenido de los atributos. Normalmente se utiliza también **SETF** con **MAKE-nombre,** puesto que no tiene sentido crear una estructura a la que luego no se pueda acceder. Suele ser normal utilizar un símbolo como medio de acceso.

La función constructora **MAKE-nombre,** tiene la siguiente sintaxis:

(MAKE-nombre clave-atributo$_1$ forma$_1$
clave-atributo$_2$ forma$_2$
...   )

En donde **forma$_I$** es evaluada y su valor asignado al atributo indicado por la clave **clave-atributo$_I$,** que se escribirá como: **nom-atributo$_I$.**

En el caso del ejemplo se habrían creado:

— El tipo de datos PERRO
— La función MAKE-PERRO
— Las funciones de acceso PERRO-RAZA, PERRO-EDAD y PERRO-PERTEN-A
— El predicado PERRO-P
— La función de copia COPY-PERRO

La utilización de estas funciones se mostrará con el ejemplo, en donde se crea un objeto de tipo PERRO, de nombre OSCAR.

```
(SETF OSCAR (MAKE-PERRO
      :RAZA MASTIN
      :EDAD 4
      :PERTEN-A LOLA) -> #S(PERRO RAZA MASTIN
                           EDAD 4 PERTEN-A LOLA)
```

SETF retorna el resultado de evaluar MAKE-PERRO, que retorna el contenido de los atributos de la ocurrencia OSCAR de la estructura PERRO expresados tal como aparece en el ejemplo.

```
(SETF (PERRO-EDAD OSCAR) 5) -> 5      Con las funciones de
                                      acceso puede modifi-
                                      carse el contenido
                                      de los atributos de
                                      OSCAR.
(PERRO-EDAD OSCAR) -> 5
(PERRO-P OSCAR) -> T
(SETF CHUCHO (MAKE-PERRO) -> #S(PERRO RAZA DESCONOCIDA
                    EDAD DESCONOCIDA PERTEN-A NADIE)
(PERRO-RAZA CHUCHO)  -> DESCONOCIDA
```

Si no se especifican los contenidos de los atributos MAKE-PERRO asigna los valores señalados por omisión.

CAPITULO XII

OPERACIONES DE ENTRADA Y SALIDA

En este capítulo se presentarán los principales medios que tiene LISP para interaccionar con el exterior. Como ya se ha comentado en otro lugar de esta obra, el funcionamiento de LISP se basa en el bucle READ-EVAL-PRINT. Las expresiones, introducidas desde el teclado u otro dispositivo de entrada, son leídas e interpretadas, posteriormente se evalúan y finalmente se muestra su valor, típicamente en pantalla. En las páginas que siguen se hablará de los procedimientos para que un programa reciba datos de cualquier dispositivo externo o se los envíe.

Canales de entrada y salida

Los **canales** son objetos LISP que sirven como fuentes y/o sumideros de datos. A través de ellos se produce el diálogo entre el intérprete LISP y el mundo exterior. Normalmente existe un canal de entrada/salida por defecto, que está unido a la consola: el usuario puede definir otros, ligados a impresoras, ficheros en disco, etc. Puede cambiarse el canal por defecto, con lo cual las operaciones básicas se realizarán a través de otro dispositivo externo diferente de la consola.

Frecuentemente, los canales están unidos a un fichero del que pueden leerse o en el que pueden escribirse datos. Las operaciones básicas que pueden realizarse con canales y ficheros son la apertura y el cierre.

En al **apertura** de un fichero se crea un canal unido a él. La sintaxis de una sentencia de apertura es la siguiente:

(SETF nom-str (OPEN nom-fich: DIRECTION: clave))

- nom-str: Es el nombre que se asigna al canal que va a estar unido al fichero que se está abriendo.
- nom-fich: Indica el **camino** hacia el fichero, incluyendo su nombre. La forma de describir este camino varía de un sistema operativo a otro. Típicamente suele ser una cadena de caracteres (escrita, por tanto, entre comillas) que indica la unidad de disco y el camino a través del árbol de directorios para llegar al fichero. En un sistema que utilice el DOS, un camino podría ser: "b:/directorio/fichero.dat".
- :DIRECTION :CLAVE: Es un modificador; especifica si el canal que se va a construir es de entrada, salida o ambas cosas. El valor de **:clave** sería, respectivamente, **:input, :output** o **:io.** Además de **:DIRECTION** pueden existir otros modificadores que no se van a tratar aquí.

Una vez evaluada esta sentencia, **nom-str** queda ligado al canal que conduce al fichero **nom-fich;** en ciertas instrucciones de entrada y salida constará **nom-str,** para indicar el destino o la procedencia de los datos.

Cuando finalizan las operaciones con un fichero es necesario cerrar el canal que conduce a él. Para esto se emplea CLOSE, cuya sintaxis es:

(CLOSE nom-str)

La evaluación de esta expresión produce el cierre del canal nombrado por **nom-str.** A partir de entonces no se podrán llevar a cabo operaciones de entrada y salida a través de ese canal.

Entrada de datos

La forma básica de leer datos es el empleo de la función READ. Su sintaxis es:

(READ nom-str)

READ lee un objeto LISP del canal señalado por **nom-str.** Si se escribe solamente **(READ),** la lectura se realiza del canal por defecto, normalmente el teclado de la consola. En este último caso, la evaluación del programa se detendría hasta que el usuario introdujera el objeto LISP en cuestión y pulsara RETORNO. La expresión **(READ nom-str)** retorna como valor el objeto leído, por lo que puede ser suministrada como argumento a una función o forma especial. Un empleo frecuente es:

(SETF nom-var (READ nom-str))

donde el valor leído es asignado a la variable identificada por **nom-var.**

Otra función de entrada/salida particularmente útil es YES-OR-NO-P. Su sintaxis es:

(YES-OR-NO-P cadena)

Esta expresión muestra **cadena** en la pantalla y espera a que el usuario teclee "yes" o "no". Evalúa a NIL si se responde "no" y a T si se contesta "yes". Es muy útil para emplearla en construcciones de la forma:

(IF (YES-OR-NO-P cadena) acción$_1$ acción$_2$)

con lo cual se consigue la transferencia del control a **acción$_1$** o a **acción$_2$** a voluntad.

Salida de datos sin formato

WRITE es la función básica de salida. Su sintaxis es:

(WRITE objeto :STREAM nom-str)

- objeto: es un objeto LISP al que se va a dar salida por el canal especificado. Previamente, como cualquier argumento de función, es evaluado.
- nom-str: identifica el canal de salida seleccionado. El conjunto **:STREAM nom-str** puede omitirse; en ese caso se utiliza el canal por defecto.

WRITE vale **objeto.** Esta función puede llevar una serie de modificadores que especifican cómo debe hacerse la salida. Por ejemplo, cuántos niveles de paréntesis han de escribirse en caso de que **objeto** sea un conjunto de listas anidadas. También puede detallarse el empleo de mayúsculas o minúsculas, el número de elementos de un determinado nivel de paréntesis que se imprimirán (los que quedan sin imprimir se sustituyen por puntos sus-

pensivos), la base empleada para escribir los números (sistema decimal, binario, etc), etc.

Existen una serie de funciones que surgen de la particularización de WRITE para unos determinados modificadores. Todas tienen la misma sintaxis:

(*PRIN* objeto nom-str)

y escriben **objeto** por el canal **nom-str** o por el canal por defecto, si se omite **nom-str.**

PRIN puede ser:

- PRIN1: Escribe la representación impresa normal de **objeto.** Puede decirse que la salida de PRIN1 puede usarse de entrada para READ.
- PRINT: Es idéntico a PRIN1, pero la representación de **objeto** es precedida de una señal de **NUEVA LINEA** y seguida de un espacio.
- PPRINT: La escritura se realiza especialmente formateada para una mejor comprensión por el usuario.
- PRINC: Omite caracteres de control como la barra invertida (\), las comillas (" "), etc. Produce la salida más elegante, pero no puede ser usada como entrada para READ.

TERPRI es una función que envía una señal de **NUEVA LINEA** al canal de salida deseado.

Todas las funciones de salida retornan **objeto** excepto PPRINT. Por tanto, si se usan para enviar datos a la pantalla, como en

(PRIN1 'FOO) → FOO FOO

el dato en cuestión aparece dos veces: una debida al efecto lateral del PRIN1 (que es la escritura) y otra debida al efecto principal de la evaluación de toda expresión LISP, que consiste en mostrar su valor al canal por defecto. Si las funciones de salida se emplean en el cuerpo de una función o en cualquier estructura de tipo PROGN implícita (siempre que sea en posición distinta de la última) sus valores no se retornan, por lo que sólo se advertiría el efecto lateral.

Salidas con formato

La función FORMAT es el procedimiento más elaborado de salida de datos. Permite la realización de salidas con formatos

definidos por el usuario, a semejanza de instrucciones FORTRAN o BASIC. Su sintaxis es la siguiente:

(FORMAT destino cadena argumentos)

Donde:

- destino: Puede ser un símbolo ligado a un canal de salida o NIL. En el primer caso los datos son enviados por dicho canal en forma de cadena de caracteres (la *cadena de salida)* y FORMAT retorna NIL. En caso de que **destino** sea NIL FORMAT retorna la cadena de salida sin enviarla a ningún canal.
- cadena: Es una cadena de caracteres, escrita, por tanto, entre comillas. Son enviados a **destino** todos los caracteres de esta cadena, excepto aquellos que van precedidos del signo "~", que se consideran *directivas de formato.*
- argumentos: Cada vez que aparece una directiva de formato se toma uno o varios elementos de **argumentos** y se los introduce en la cadena de salida con el formato indicado por la directiva.

FORMAT tiene como efecto la creación de una cadena de salida a partir de **cadena** y de **argumentos,** dando a estos últimos el formato especificado por las *directivas.* No se puede hacer aquí una exposición exhaustiva de las posibilidades de esta función, pero el lector puede hacerse una idea con los siguientes ejemplos.

```
(FORMAT NIL "FOO") -> "FOO"

(SETQ X 7) -> 7

(SETQ Y 9) -> 9

(FORMAT NIL "Las soluciones son ~A y ~A" X Y) ->

-> "Las soluciones son 7 y 9"
```

En este último caso se incorporan a la cadena de salida todos los caracteres de **cadena** excepto "~A". Esta es una directiva que ordena imprimir un argumento sin caracteres de control, como hacía PRINC. El primer argumento es "X", y es éste el que se toma. Si hay varias directivas y varios argumentos, cada una de ellas se encarga de uno o varios de éstos, tomándolos de izquierda a derecha. Existen directivas que definen el formato de los números, el sistema de numeración que se emplea para representarlos, o que introducen la orden de RETORNO, caracteres separadores, etc.

Carga de ficheros

La función LOAD permite cargar en memoria ficheros contenidos en un disco. Todas las expresiones que haya en esos ficheros son leídas e interpretadas, y retornados sus valores como si fueran tecleadas por el usuario. Gracias a esta función pueden crearse procedimientos en un editor de textos, aprovechando las ventajas de una de estas herramientas, y trabajar con ellos en un entorno LISP posteriormente.

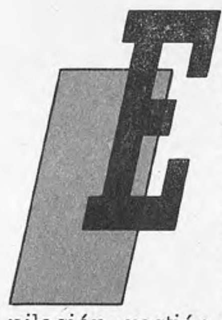
La sintaxis de LOAD es:

(LOAD camino)

donde **camino** es la identificación de un fichero en el árbol de directorios. Ha de ser una cadena de caracteres escrita, por tanto, entre comillas.

CAPITULO XIII

EL ENTORNO LISP



El entorno LISP es todo el conjunto de programas que hacen posible al usuario la creación, ejecución y corrección de programas LISP. Está compuesto principalmente por un intérprete que lleva a cabo el bucle READ-EVAL-PRINT. Desde el intérprete se tiene acceso normalmente a un editor, que ayuda en la creación de los programas. Pueden existir también una serie de funciones auxiliares que permiten la depuración, compilación, gestión de memoria, etc.

El intérprete LISP

La base del entorno LISP es un intérprete, que puede trabajar en varios niveles. Al iniciarse el trabajo, se parte del nivel superior **(Top-Level),** identificado por una *señal de espera* (prompt), típicamente un asterisco. El usuario puede comenzar a teclear programas LISP o cargarlos desde una unidad de almacenamiento. Las expresiones, cuando están completas, son leídas, interpretadas y evaluadas mediante el bucle ya citado. En la pantalla aparecen los valores retornados y la señal identificadora del sistema, que nos da paso para introducir nuevas expresiones.

Cuando se produce un error o una interrupción, el intérprete entra en el nivel siguiente, que permite la interacción con el usuario. Este nivel está identificado por una señal de espera que contiene un 1. El usuario puede pedir información sobre el estado del proceso, pues las variables que estuvieran ligadas en el instante de la interrupción siguen estándolo. Esto facilita el proceso de de-

tección y corrección del error. Además, existen herramientas de depuración que facilitan esta tarea.

El usuario puede ordenar la entrada del proceso en un nivel inferior por medio de la función

(BREAK)

La evaluación de esta expresión produce la detención del proceso y el paso al nivel inmediatamante inferior. El comportamiento del sistema es el mismo que cuando se produce un error.

En la figura 1 se muestra una pantalla de uno de los intérpretes LISP disponibles en el mercado para ordenadores personales.

```
Copyright (C) 1985 by Gold Hill Computers

; Reading file INIT.LSP

Initialization file loaded.
Type Alt-H for help
Top-Level
* (setq x 'foo)
FOO
* x

FOO
* (setq x foo)

ERROR:
Unbound variable: FOO
1>
Back to: Top-Level
* (break)

BREAK, (CONTINUE) to continue.

1> (continue)
NIL
*
```

Figura 1.—Pantalla de un intérprete LISP para ordenador personal. Puede verse la señal de espera del nivel superior y la introducción de expresiones correctas y erróneas.

Puede verse la señal de espera del nivel superior, la introducción de una función correcta y de otra errónea, que da paso al nivel 1.

Depuración

En LISP existen varias herramientas de depuración, que ayudan al usuario en la corrección de errores en los programas. Permiten la ejecución paso a paso de éstos y la obtención de información sobre la marcha de los procesos.

Las principales herramientas de depuración de programas en LISP son TRACE, su opuesta (UNTRACE) y STEP.

(TRACE función$_1$ ... función$_N$)

Produce el efecto de reflejar en la pantalla todas las entradas y salidas de **función$_1$ ... función$_N$**, con los argumentos empleados. Ejemplos de esta macro se pueden observar en el capítulo 10.

(UNTRACE función$_1$ ... función$_N$)

Desactiva TRACE para **función$_1$ ... función$_N$**. En caso de que se llame a UNTRACE sin argumentos se considera que debe desactivarse TRACE para todas las funciones.

STEP permite ejecutar un procedimiento paso a paso con distintos niveles de detalle, controlados por el usuario de forma interactiva. Depende mucho de la versión del intérprete de que se trate, por lo que no ampliaremos aquí la descripción de esta función.

Ayuda a la depuración la introducción de comentarios e instrucciones en los propios programas. En LISP también existe la posibilidad de introducir comentarios en los procedimientos (como el REM de BASIC) mediante el punto y coma (;). El intérprete ignorará todo lo que encuentre escrito a la derecha del punto y coma dentro de una línea.

La función APROPOS permite obtener los nombres de los símbolos que contengan una determinada cadena de caracteres.

(APROPOS cadena-de-caracteres)

Busca los símbolos definidos que posean en su nombre **cadena-de-caracteres** y los muestra en la pantalla. No es puramente una herramienta de depuración, pero ayuda a evitar que un mismo nombre se emplee inadvertidamente para dos misiones diferentes.

Edición

Desde el intérprete LISP se puede acceder directamente a un editor por medio de la función

(ED camino)

donde **camino** es una cadena de caracteres que especifica el camino de acceso a través del árbol de directorios al fichero que se quiere editar.

Esta función permite acceder al editor, cargando en él el contenido del fichero. Si no se especifica **camino** también se accede al editor, pero en el estado en que quedó tras la última sesión de edición. Las características concretas del editor dependen de la firma proveedora del intérprete. Generalmente suelen estar escritos en LISP y tienen facilidades diseñadas para la edición de programas en ese lenguaje. Uno de los más conocidos es EMACS, que ha estado muy ligado a COMMON LISP.

Algunas de las características más importantes de estos editores son la posibilidad de evaluar expresiones LISP editadas en ellos sin abandonar el editor y volver al intérprete, proveer a las expresiones de una indentación normalizada que ayuda a su comprensión y ayudas al manejo de los paréntesis.

El empleo de la indentación es particularmente importante en LISP. Compárense estos dos ejemplos, que muestran dos formas de escribir una misma función:

```
(COND ((EQUAL 3 (* N 2))(CAR '(A B C)))(T (CDR '(B C A))))
```

```
(COND ((EQUAL 3 (*N 2)) (CAR '(A B C)))
      (     T           (CDR '(B C A))))
```

Compilador

Para acelerar la evaluación de los procedimientos LISP, éstos se pueden compilar, produciendo código directamente utilizable por el ordenador.

Relacionadas con esta posibilidad existen tres funciones: COMPILE, COMPILE-FILE y DISASSEMBLE.

(COMPILE función)

Se aplica a funciones LISP ya definidas y conocidas por el intérprete. Produce una función compilada y la liga al mismo nombre que previamente tenía. Lógicamente, **función** ha de ir precedido de apóstrofe para inhibir su evaluación.

(COMPILE-FILE fichero)

Produce código compilado de los procedimientos LISP contenidos en **fichero,** donde **fichero,** es un almacenamiento externo al intérprete LISP. Por tanto, **fichero** ha de ser un camino de acceso a través del árbol de directorios, entrecomillado. Ese código compilado ha de ser cargado mediante la función LOAD.

(DISASSEMBLE función)

Efectúa la operación inversa de COMPILE: partiendo de **función,** ya compilada, produce los procedimientos LISP en el código inteligible al programador.

Gestión de memoria

Los procedimientos LISP hacen un uso intensivo de la memoria en el trabajo normal. Recuérdese que muchas de las funciones de manejo de listas crean copias de sus argumentos para después modificarlas; además, las llamadas a las funciones producen el establecimiento de nuevas ligaduras y la reserva de posiciones de memoria que, al finalizar la evaluación, no serán ya accesibles.

Se produciría rápidamente el colapso del ordenador si no se reorganizarán las células y posiciones de memoria no usadas y las que han sido utilizadas, pero ya no lo son. Esta tarea la lleva a cabo un procedimiento llamado GARBAGE-COLLECTOR que se activa cada vez que la memoria se ocupa por encima de cierto umbral.

El sistema lleva la cuenta de las posiciones de memoria libres; puede decirse que mantiene una relación de células libres. Cuando la memoria se llena lo suficiente, el procedimiento trabaja en dos fases:

1.ª Confecciona una tabla con todos los objetos LISP activos en ese momento y las posiciones de memoria que ocupan.
2.ª Añade todas las posiciones no utilizadas, es decir, que no figuran en la tabla anterior, a la lista de células libres.

Así, pues, puede decirse que el GARBAGE-COLLECTOR realiza una limpieza de la memoria, eliminando posiciones que ya no son útiles.

El usuario también tiene cierto control sobre el GARBAGE-COLLECTOR mediante la función:

(COLLECT)

que lo invoca sin necesidad de que se sobrepase el umbral antes citado.

Recomendaciones generales

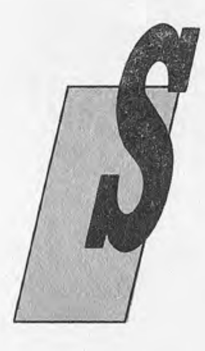
Es recomendable que el usuario trate de restringir el empleo de ligaduras globales, que siempre permanecen ocupando memoria. Resulta más conveniente y acorde con la filosofía LISP el empleo de las ligaduras locales, que se suprimen cuando ya no son necesarias. De acuerdo con esto es una buena práctica el empleo de las funciones MAKUNBOUND y FMAKUNBOUND, que eliminan, respectivamente, las ligaduras de una variable y de una función cuando ya no son útiles.

La organización preferible de un programa LISP está formada por un cierto número de pequeñas funciones que se llaman unas a otras y que resuelven aspectos parciales del problema. Esto hace los programas más inteligibles, y facilita su depuración.

Por último,. hay que insistir una vez más en que debe huirse del empleo de las construcciones PROG. En caso de que sea conveniente el uso de la iteración ha de recurrirse al DO.

CAPITULO XIV

NOTAS FINALES



uponiendo ya al lector en posesión de ciertos conocimientos básicos sobre Common LISP, el presente capítulo pretende justificar el empleo de este lenguaje, mostrando algunas de sus aplicaciones y comentando sus principales ventajas e inconvenientes.

Common LISP en el mundo LISP

Como ya se señaló en el capítulo 1, no existe una versión única del lenguaje LIDP, sino multitud de dialectos cuyas incompatibilidades son sustanciales. Podría apuntarse como causa de este hecho la dispersión de los focos en donde ha madurado este lenguaje, normalmente radicados en los laboratorios de investigación en Inteligencia Artificial. En cada uno de ellos se tendía a mejorar el lenguaje adaptándolo a las necesidades propias. Esto, unido a la falta de primacía clara de unos dialectos sobre otros y a que todavía no ha terminado la evolución de los mismos, hace que surjan diferentes tipos de LISP.

La versión de LISP de aceptación más generalizada fue el LISP 1.5, un dialecto del año 1962. En la actualidad está en desuso por carecer de la potencia que tienen las versiones modernas del lenguaje. Los dialectos más extendidos hoy en día son: MacLISP, Interlisp, UCI LISP y Lisp Machine LISP.

Common LISP surge como un intento de establecer una unificación del lenguaje en el momento en que comienzan a diverger los sucesores de MacLISP debido a los diferentes entornos a

los que deben adaptarse (ordenadores personales, ordenadores de tiempo compartido y superordenadores). Por tanto, puede decirse que Common LISP se basa en la herencia de MacLISP, aunque tiene gran influencia de ZetaLISP (un dialecto surgido de MacLISP para ordenadores personales).

Desgraciadamente, seguirán manteniéndose algunas incompatibilidades entre las diferentes implantaciones de Common LISP, ya que éstas vienen forzadas por cada entorno de trabajo. Sin embargo, con este lenguaje se ha pretendido tener un dialecto básico común que excluya los elementos que no puedan adaptarse a la mayoría de las máquinas, pero que permita, por otra parte, una implantación sencilla de las características adicionales que se juzguen necesarias.

Las especificaciones de Common LISP fueron establecidas por consenso entre un gran número de personas procedentes de diversas instituciones (están recogidas en el Manual de Referencia de Common LISP, escrito por Guy L. Steele); precisamente en eso se funda la idea de normalización del lenguaje con que ha surgido. Las modificaciones y actualizaciones del mismo se pretende que sean lentas y realizadas tras una cuidadosa elaboración. Como campo de ensayo de las mismas servirán las diferentes implantaciones que surjan de este lenguaje.

En cuanto a su relación con otros dialectos puede decirse que, en general, trata de ser compatible con ZetaLISP, MacLISP e Interlisp, aproximadamente en este orden. La fuente de incompatibilidades puede estar no sólo en los diferentes conjuntos de primitivas que ofrece cada dialecto o en su diferente nomenclatura; esto sería fácilmente subsanable por el usuario debido a la posibilidad de definir de forma sencilla las funciones necesarias. Las incompatibilidades alcanzan a veces los tipos de objetos permitidos o cuestiones tan básicas como el valor del CAR o del CDR de NIL.

La capacidad de cálculo depende en gran medida de la implantación. Como ya se comentó en el primer capítulo, las versiones destinadas a pequeños ordenadores todavía no son muy potentes. Sin embargo, hay versiones como la S-1LISP que poseen un compilador que genera código para cálculo numérico competitivo en velocidad con el obtenido de un compilador FORTRAN.

Common LISP está también especialmente preparado para soportar herramientas de ayuda a la construcción de aplicaciones.

Esta visión general puede dar la impresión de un gran desorden dentro del mundo LISP. Sin embargo, esta imagen negativa no tiene razón de ser, puesto que Common LISP está apoyado por los principales centros de investigación en Inteligencia Artificial, desde universidades a fabricantes de ordenadores, y por las más importantes firmas productoras de programas. Esta conjunción de esfuerzos está preparando las bases para un rápido desarrollo en

un futuro próximo, en el que se dispondrá de ordenadores de mayor capacidad de memoria y velocidad de proceso.

Desarrollos basados en LISP. Máquinas LISP

En los últimos tiempos ha aparecido en el mercado un número creciente de productos basados en técnicas de Inteligencia Artificial.

Cuando se trata de desarrollar una de estas aplicaciones hay dos posibilidades: partir de cero y construir todo el sistema apoyándose en un lenguaje de programación o bien tratar de aprovechar parcialmente otros desarrollos previos en el campo donde se pretenda trabajar.

Evidentemente, la primera opción resulta más laboriosa, pero en muchas ocasiones es la única alternativa por la inexistencia o inadaptabilidad de los programas existentes. Entre los lenguajes de programación, la elección de uno u otro depende de la forma en que se desee llegar al objetivo. En rigor, serviría cualquier lenguaje de alto nivel, pero parece más apropiado utilizar alguno de los más extendidos en Inteligenncia Artificial por su flexibilidad. Entre ellos LISP tiene un puesto destacado.

Un entorno de programación LISP, como el proporcionado por ordenadores específicos (las *máquinas LISP*), incluye una serie de facilidades que lo convierten en una de las mejores opciones para los desarrollos en Inteligencia Artificial. Estos entornos están dotados de una extensa biblioteca de funciones, editores específicos para LISP, herramientas de depuración, compiladores, sistemas de ventanas, etc., que, además, suelen estar integrados. Esto último permite que se interrelacionen para ahorrar tiempo y trabajo en el desarrollo de las aplicaciones.

Los fabricantes de equipos informáticos están lanzando al mercado diversas máquinas de este tipo. En 1981 aparecieron las primeras. Xerox, con sus sistema Dolphin/1100, comercializado a mediados de ese año; Symbolics Inc., productor de LM2, y Lisp Machine Inc., fabricante de Lisp Machine, fueron los primeros en ver el prometedor futuro de este campo. A partir de éstos han surgido nuevos equipos, más evolucionados, como el sistema Lambda, de Lisp Machine, la máquina Symbolic 3600, de Symbolics, o el Explorer de Texas Instruments y Sperry.

Los usos de las máquinas LISP son muy diversos. Pueden ir desde el diseño de vídeo-juegos hasta desarrollos de visión artificial o enseñanza asistida por ordenador. En todos ellos se detecta el fenómeno común del aumento de productividad en cuanto a la programación se refiere. La tendencia natural de estas máquinas será, por tanto, la integración con los sitemas tradicionales de

ordenadores, lo que permitirá a éstos el desarrollo de productos relacionados con la Inteligencia Artificial, así como otras aplicaciones más convencionales.

En caso de que no se considere adecuado el desarrollo de una aplicación partiendo de cero se dispone también de una serie de herramientas auxiliares. El número y calidad de estas herramientas es particularmente importante en el campo de los Sistemas Expertos, uno de los más fructíferos de la Inteligencia Artificial (remitimos al lector al volumen de la B. B. I. dedicado a la I. A. y los S. E.)

Conviene recordar que un Sistema Experto pretende simular la actuación de un experto humano en un tema concreto. Para ello debe tener almacenados todos los conocimientos del experto sobre ese tema y debe saber interpretarlos y sacar consecuencias de los mismos. Por lo tanto, su arquitectura típica está formada, a grandes rasgos, por una Base de Conocimientos y un Motor de Inferencias. En aquélla se almacenan, de diversas formas, los conocimientos propios del experto. El Motor de Inferencias es el programa que aplica el contenido de la Base de Conocimientos a la resolución de los problemas planteados.

Hay dos tipos de herramientas de ayuda a la creación de Sistemas Expertos: los lenguajes de representación del conocimiento y los llamados sistemas concha. Los primeros son lenguajes de alto nivel y de propósito general. Han sido desarrollados específicamente para aplicaciones de ingeniería del conocimiento, aunque pueden ser útiles para una amplia variedad de campos. En general están construidos en LISP, con lo que tienen sus mismos defectos de "ineficiencia" y gozan también de todas sus ventajas de flexibilidad. Los más conocidos son ROSIE, OPS-5, RLL y HEARSAY-III. Todos ellos ayudan a la construcción de Bases de Conocimiento, proporcionando la estructura básica para almacenamiento de información. Incluyen una serie de estructuras de control que pueden ser consideradas como Motor de Inferencias, pero gran parte de las tareas de control quedan en manos del usuario.

Respecto a los sistemas concha puede decirse que son sistemas expertos en los que se ha eliminado el contenido de su Base de Conocimientos. Sólo queda el Motor de Inferencias y la estructura donde se almacenan los datos. De aquí su nombre, que pretende recoger la idea de una estructura vacía. Con ellos se trabaja a un nivel más alto que el de la programación. Se ahorra, pues, la mayor parte del esfuerzo necesario para crear la aplicación, pero como contrapartida se produce una pérdida de flexibilidad, que se hace notoria cuando el campo en el que se trabaja no se asemeja demasiado al problema para el que se desarrolló la concha. Entre los más conocidos están EMYCIN (derivado de MYCIN, sistema utilizado para el diagnóstico de enfermedades infecciosas en la sangre), KAS (creado a partir de PROSPECTOR, que interpreta datos geológicos y ha sido utilizado para descubrir yacimientos de minerales) y EXPERT (derivado de CASNET, que hace diagnósitco y terapia del glaucoma).

LISP: ventajas e inconvenientes

Al llegar a este punto puede parecer ocioso dedicar un apartado a juzgar las ventajas e inconvenientes del LISP cuando las principales han sido ya señaladas anteriormente. Sin embargo, es interesante analizarlas con más detenimiento.

Para comenzar, conviene dejar bien claro que, en lo referente a eficiencia de ejecución y de utilización de recursos, el lenguaje de programación LISP sigue a la zaga de otros menos flexibles. Lo que hace que la balanza pueda terminar volcándose de su lado es el descenso del precio de los equipos informáticos y la mayor incidencia en el coste total de un proyecto del tiempo dedicado al desarrollo y mantenimiento de los programas. Dicho de otro modo, para hacer que un programa escrito en LISP sea tan efectivo como un programa con el mismo cometido, escrito en FORTRAN, deben utilizarse equipos más caros. Sin embargo, este gasto extra en material se ve compensado con creces por el ahorro en el trabajo de desarrollo del programa.

La flexibilidad de LISP radica en el almacenamiento en memoria de los datos y programas como elementos no secuenciales unidos por punteros. FORTRAN y otros lenguajes, por el contrario, almacenan sus datos e instrucciones en posiciones de memoria secuenciales. Esto conlleva que tras añadir una línea a un programa, un elemento a una matriz, o, simplemente, modificar el tipo de una variable, debe compilarse y montarse de nuevo todo el programa. Si se trata de un programa de tamaño medio esto puede tardar algunos minutos. Sin embargo, en LISP, para hacer una modificación similar, el intérprete sólo debe cambiar unos cuantos punteros y añadir algunas células constructivas, operaciones que se hacen casi instantáneamente.

La mayor flexibilidad se cobra su precio en espacio de memoria y tiempo de ejecución: los programas en LISP ocupan el doble que en FORTRAN. En este lenguaje, la velocidad se ve muy favorecida por el almacenamiento secuencial. Para acceder a una determinada posición basta con situarse al comienzo del bloque que la contiene y buscar dentro de él. En LISP deben recorrerse muchos niveles de punteros hasta llegar a la posición buscada, con lo que la velocidad viene a reducirse a la mitad.

En las versiones modernas de LISP se utilizan dos técnicas para paliar estos defectos. Un primer método consiste en repre-

sentar las instrucciones y direcciones de memoria con un mayor número de bits. Esto permite incorporar en cada instrucción información que facilite la localización de la siguiente.

Otra técnica emplea secciones contiguas de memoria para almacenar los elementos que así lo requieran, por ejemplo las matrices. De este modo se facilita el acceso a su contenido partiendo de su primer componente. Las características generales de la memoria no quedan modificadas, porque estos elementos están unidos a los demás sólo por un puntero.

Otras características de flexibilidad que restan eficiencia son la definición del tipo de las variables y la asignación de memoria para ellas durante la ejecución. En lenguajes compilados como FORTRAN o PASCAL, el compilador comprueba que los tipos de datos son los adecuados para las operaciones que van a realizarse con ellos y les asigna la cantidad de memoria que se indica en su declaración de tipo o de dimensiones. LISP realiza estas tareas durante la ejecución, por lo que invierte gran cantidad de tiempo en multitud de comprobaciones. De nuevo la solución a esto es la modificación del equipo. Si se representan los datos con más bits se dispone de espacio adicional para almacenar el tipo, con lo cual la comprobación es simultánea al acceso a la información.

LISP presenta ciertas particularidades en lo tocante a la representación de la información. Al ser la misma para programas y datos, los programas pueden modificarse a sí mismos o crear otros programas. Estos pueden, incluso, compilarse sin perjudicar el funcionamiento del conjunto.

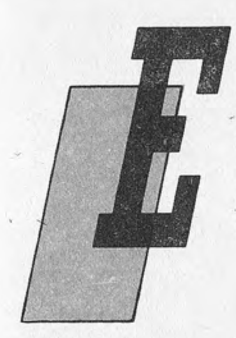
El eficiente manejo de los símbolos facilita enormemente el diseño de programas que se relacionen con el mundo real. Si se trata de una aplicación en el campo de la electrónica, por ejemplo, es posible definir como tipo de datos un transistor, escribiendo una función que simule su comportamiento y describa sus características principales. Una vez hecho esto es posible olvidar su representación interna al trabajar con este transistor abstracto. Es posible definir otros objetos partiendo de los ya existentes; así se podría simular algún circuito electrónico cuyos componentes fueran, entros, elementos del tipo TRANSISTOR.

Para terminar, debe destacarse la característica interactiva del lenguaje, que ofrece una alternativa al método convencional de programación. En FORTRAN, por ejemplo, lo normal es escribir todo un programa o, por lo menos, un módulo del mismo, antes de probarlo y depurarlo. Una vez probado suele ser necesario modificarlo, compilarlo y montarlo de nuevo, para probar los cambios introducidos. En LISP, sin embargo, lo normal es empezar a escribir pequeñas funciones que van siendo probadas y modificadas en el momento de su creación. Según se avanza en la concepción del programa se van usando los procedimientos creados en las primeras fases. El programador en LISP tiene, por tanto, mayor libertad para experimentar y crear.

En resumen, puede decirse que, a pesar de su ineficiencia en el manejo de los recursos del ordenador, LISP permite desarrollar con mayor eficacia los recursos creativos del programador humano, y esto es lo que a medio plazo tendrá más peso a la hora de elegir un lenguaje para el desarrollo de una aplicación informática.

APENDICE A

DICCIONARIO DE LOS TERMINOS INGLESES MAS FRECUENTES EN LA BIBLIOGRAFIA SOBRE LISP



En esta obra se ha hecho un especial esfuerzo por evitar el empleo de terminología anglosajona. Para facilitar el manejo de la bibliografía recomendada se incluye aquí un breve diccionario de referencia con los términos más frecuentes e importantes.

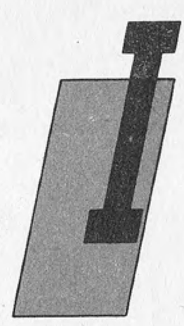
Atributo	Slot
Bucle	Loop
Cadena de caracteres	String
Camino hacia el fichero	Pathname
Canales	Stream
Compilar	To compile
Célula constructiva (o elemental)	Cons cell
Forma especial	Special form
Implantación	Implementation
Ligadura	Binding
Lista de asociación*	Association list
Lista de propiedades	Property list
Marca (etiqueta o señal)**	Tag
Matriz	Array

* También ha sido citada en la forma de empleo de listas como tablas.
** Utilizada en las expresiones del tipo (GO marca).

Montar	To link
Nivel superior	Top level
Puntero	Pointer
Señal de espera	Prompt
Variable no ligada	Unbound variable

APENDICE B

LOCALIZACION DE LAS FUNCIONES, FORMAS ESPECIALES Y MACROS



Incluimos a continuación el índice alfabético de las funciones, formas especiales y macros empleadas en esta obra, con la indicación del capítulo en que se introducen.

BIBLIOGRAFIA

The programming language LISP: its operation and applications.
Berkeley Bobrow. The M. I. T. press, 1974.

Artificial Intelligence programming.
Charniak Riesbek McDermott. L. E. A. Publishers, 1980.

LISP: A gentle introduction to symbolic computation.
Touretzky. Harper and Row Publishers, 1984.

Common LISP: The Language.
Steele. Digital Press, 1984.

LISP.
Winston Horn. Addison-Wesley, 1984.

LISP software-development environments increase programmer productivity.
Marrin. Revista EDN, 23 agosto 1984.



Aunque es uno de los lenguajes de programación más antiguos (finales de los cincuenta) es ahora cuando LISP comienza a despertar verdadero furor en el mundo de la informática. La razón estriba fundamentalmente en sus aplicaciones en el campo de la Inteligencia Artificial, el abaratamiento de los potentes equipos que precisa y el creciente coste del tiempo de programación.

Mucho más flexible que lenguajes de alto nivel tan conocidos como BASIC, FORTRAN o PASCAL es, además, sumamente fácil de aprender, con un característico uso de los paréntesis.

En esta obra pretendemos adentrarle en el mundo del LISP presentándole toda la teoría necesaria arropada por constantes y precisos ejemplos, escritos en COMMOM LISP, dialecto que se está perfilando como el estándar de este lenguaje.

395 pts.
(incluido IVA)